TraitsUI 4 User Manual

Release 4.6.0.dev153

Enthought, Inc.

August 18, 2015

Contents

1 TraitsUI 4 User Manual **1**
- 1.1 TraitsUI 4 User Manual . . . 1
- 1.2 Introduction . . . 2
- 1.3 The View and Its Building Blocks . . . 4
- 1.4 Customizing a View . . . 13
- 1.5 Advanced View Concepts . . . 19
- 1.6 Controlling the Interface: the Handler . . . 26
- 1.7 TraitsUI Themes . . . 32
- 1.8 Introduction to Trait Editor Factories . . . 37
- 1.9 The Predefined Trait Editor Factories . . . 43
- 1.10 Advanced Trait Editors . . . 70
- 1.11 "Extra" Trait Editor Factories . . . 89
- 1.12 Tips, Tricks and Gotchas . . . 96
- 1.13 Appendix I: Glossary of Terms . . . 97
- 1.14 Appendix II: Editor Factories for Predefined Traits . . . 98

2 TraitsUI 4 Tutorials **101**
- 2.1 Writing a graphical application for scientific programming using TraitsUI 4 . . . 101

3 TraitsUI 4 Demos **121**
- 3.1 Standard Editors . . . 121
- 3.2 Advanced Demos . . . 122

4 Indices and tables **125**

CHAPTER 1

TraitsUI 4 User Manual

1.1 TraitsUI 4 User Manual

Authors Lyn Pierce, Janet Swisher

Version Document Version 4

Copyright 2005, 2008 Enthought, Inc. All Rights Reserved.

Redistribution and use of this document in source and derived forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source or derived format (for example, Portable Document Format or Hypertext Markup Language) must retain the above copyright notice, this list of conditions and the following disclaimer.
- Neither the name of Enthought, Inc., nor the names of contributors may be used to endorse or promote products derived from this document without specific prior written permission.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

All trademarks and registered trademarks are the property of their respective owners.

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin TX 78701
1.512.536.1057 (voice)
1.512.536.1059 (fax)
http://www.enthought.com
info@enthought.com

1.2 Introduction

This guide is designed to act as a conceptual guide to *TraitsUI*, an open-source package built and maintained by Enthought, Inc. The TraitsUI package is a set of GUI (Graphical User Interface) tools designed to complement *Traits*, another Enthought open-source package that provides explicit typing, validation, and change notification for Python. This guide is intended for readers who are already moderately familiar with Traits; those who are not may wish to refer to the Traits User Manual for an introduction. This guide discusses many but not all features of TraitsUI. For complete details of the TraitsUI API, refer to the *Traits API Reference*.

1.2.1 The Model-View-Controller (MVC) Design Pattern

A common and well-tested approach to building end-user applications is the *MVC* ("Model-View-Controller") design pattern. In essence, the MVC pattern the idea that an application should consist of three separate entities: a *model*, which manages the data, state, and internal ("business") logic of the application; one or more *view*s, which format the model data into a graphical display with which the end user can interact; and a *controller*, which manages the transfer of information between model and view so that neither needs to be directly linked to the other. In practice, particularly in simple applications, the view and controller are often so closely linked as to be almost indistinguishable, but it remains useful to think of them as distinct entities.

The three parts of the MVC pattern correspond roughly to three classes in the Traits and TraitsUI packages.

- Model: *HasTraits* class (Traits package)
- View: View class (TraitsUI package)
- Controller: *Handler* class (TraitsUI package)

The remainder of this section gives an overview of these relationships.

The Model: HasTraits Subclasses and Objects

In the context of Traits, a model consists primarily of one or more subclasses or *instance*s of the HasTraits class, whose *trait attribute*s (typed attributes as defined in Traits) represent the model data. The specifics of building such a model are outside the scope of this manual; please see the Traits User Manual for further information.

The View: View Objects

A view for a Traits-based application is an instance of a class called, conveniently enough, View. A View object is essentially a display specification for a GUI window or *panel*. Its contents are defined in terms of instances of two other classes: *Item* and *Group*. [1] These three classes are described in detail in *The View and Its Building Blocks*; for the moment, it is important to note that they are all defined independently of the model they are used to display.

Note that the terms *view* and *View* are distinct for the purposes of this document. The former refers to the component of the MVC design pattern; the latter is a TraitsUI construct.

The Controller: Handler Subclasses and Objects

The controller for a Traits-based application is defined in terms of the *Handler* class. [2] Specifically, the relationship between any given View instance and the underlying model is managed by an instance of the Handler class. For simple interfaces, the Handler can be implicit. For example, none of the examples in the first four chapters includes or requires any specific Handler code; they are managed by a default Handler that performs the basic operations of

[1] A third type of content object, Include, is discussed briefly in *Include Objects*, but presently is not commonly used.
[2] Not to be confused with the TraitHandler class of the Traits package, which enforces type validation.

window initialization, transfer of data between GUI and model, and window closing. Thus, a programmer new to TraitsUI need not be concerned with Handlers at all. Nonetheless, custom handlers can be a powerful tool for building sophisticated application interfaces, as discussed in *Controlling the Interface: the Handler*.

1.2.2 Toolkit Selection

The TraitsUI package is designed to be toolkit-independent. Programs that use TraitsUI do not need to explicitly import or call any particular GUI toolkit code unless they need some capability of the toolkit that is not provided by TraitsUI. However, *some* particular toolkit must be installed on the system in order to actually display GUI windows.

TraitsUI uses a separate package, traits.etsconfig, to determine which GUI toolkit to use. This package is also used by other Enthought packages that need GUI capabilities, so that all such packages "agree" on a single GUI toolkit per application. The etsconfig package contains a singleton object, **ETSConfig** (importable from *traits.etsconfig.api*), which has a string attribute, **toolkit**, that signifies the GUI toolkit.

The values of **ETSConfig.toolkit** that are supported by TraitsUI version 4 are:

- 'wx': wxPython, which provides Python bindings for the wxWidgets toolkit.
- 'qt4': PyQt, which provides Python bindings for the Qt framework version 4.
- 'null': A do-nothing toolkit, for situations where neither of the other toolkits is installed, but Traits is needed for non-UI purposes.

The default behavior of TraitsUI is to search for available toolkit-specific packages in the order listed, and uses the first one it finds. The programmer or the user can override this behavior in any of several ways, in the following order of precedence:

1. The program can explicitly set **ETSConfig.toolkit**. It must do this before importing from any other Enthought Tool Suite component, including traits. For example, at the beginning of a program:

```
from traits.etsconfig.api import ETSConfig
ETSConfig.toolkit = 'wx'
```

2. The user can specify a -toolkit flag on the command line of the program.
3. The user can define a value for the ETS_TOOLKIT environment variable.

Warning: The default order of toolkits will change in TraitsUI 5.0 to prefer 'qt4' over 'wx'.

1.2.3 Structure of this Manual

The intent of this guide is to present the capabilities of the TraitsUI package in usable increments, so that you can create and display gradually more sophisticated interfaces from one chapter to the next.

- *The View and Its Building Blocks*, *Customizing a View*, and *Advanced View Concepts* show how to construct and display views from the simple to the elaborate, while leaving such details as GUI logic and widget selection to system defaults.
- *Controlling the Interface: the Handler* explains how to use the Handler class to implement custom GUI behaviors, as well as menus and toolbars.
- *TraitsUI Themes* described how to customize the appearance of GUIs through *themes*.
- *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories* show how to control GUI widget selection by means of trait *editor*s.
- *Tips, Tricks and Gotchas* covers miscellaneous additional topics.

- Further reference materials, including a *Appendix I: Glossary of Terms* and an API summary for the TraitsUI classes covered in this Manual, are located in the Appendices.

1.3 The View and Its Building Blocks

A simple way to edit (or simply observe) the attribute values of a *HasTraits* object in a GUI window is to call the object's configure_traits() [3] method. This method constructs and displays a window containing editable fields for each of the object's *trait attribute*s. For example, the following sample code [4] defines the SimpleEmployee class, creates an object of that class, and constructs and displays a GUI for the object:

Example 1: Using configure_traits()

```
# configure_traits.py -- Sample code to demonstrate
#                        configure_traits()
from traits.api import HasTraits, Str, Int
import traitsui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

sam = SimpleEmployee()
sam.configure_traits()
```

Unfortunately, the resulting form simply displays the attributes of the object **sam** in alphabetical order with little formatting, which is seldom what is wanted:

Fig. 1.1: Figure 1: User interface for Example 1

[3] If the code is being run from a program that already has a GUI defined, then use edit_traits() instead of configure_traits(). These methods are discussed in more detail in *Displaying a View*.

[4] All code examples in this guide that include a file name are also available as examples in the `tutorials/doc_examples/examples` subdirectory of the Traits docs directory. You can run them individually, or view them in a tutorial program by running: **`python`** *`Traits_dir`*`/tutorials/tutor.py` *`Traits_dir`*`/docs/tutorials/doc_examples`

1.3.1 The View Object

In order to control the layout of the interface, it is necessary to define a View object. A View object is a template for a GUI window or panel. In other words, a View specifies the content and appearance of a TraitsUI window or panel display.

For example, suppose you want to construct a GUI window that shows only the first three attributes of a SimpleEmployee (e.g., because salary is confidential and the employee number should not be edited). Furthermore, you would like to specify the order in which those fields appear. You can do this by defining a View object and passing it to the configure_traits() method:

Example 2: Using configure_traits() with a View object

```
# configure_traits_view.py -- Sample code to demonstrate
#                             configure_traits()

from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item
import traitsui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str
    employee_number = Str
    salary = Int

view1 = View(Item(name = 'first_name'),
             Item(name = 'last_name'),
             Item(name = 'department'))

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting window has the desired appearance:

Fig. 1.2: Figure 2: User interface for Example 2

A View object can have a variety of attributes, which are set in the View definition, following any Group or Item objects.

The sections on *Contents of a View* through *Advanced View Concepts* explore the contents and capabilities of Views. Refer to the *Traits API Reference* for details of the View class.

Except as noted, all example code uses the configure_traits() method; a detailed description of this and other techniques for creating GUI displays from Views can be found in *Displaying a View*.

1.3.2 Contents of a View

The contents of a View are specified primarily in terms of two basic building blocks: Item objects (which, as suggested by Example 2, correspond roughly to individual trait attributes), and Group objects. A given View definition can contain one or more objects of either of these types, which are specified as arguments to the View constructor, as in the case of the three Items in Example 2.

The remainder of this chapter describes the Item and Group classes.

The Item Object

The simplest building block of a View is the *Item* object. An Item specifies a single interface *widget*, usually the display for a single trait attribute of a HasTraits object. The content, appearance, and behavior of the widget are controlled by means of the Item object's attributes, which are usually specified as keyword arguments to the Item constructor, as in the case of *name* in Example 2.

The remainder of this section describes the attributes of the Item object, grouped by categories of functionality. It is not necessary to understand all of these attributes in order to create useful Items; many of them can usually be left unspecified, as their default values are adequate for most purposes. Indeed, as demonstrated by earlier examples, simply specifying the name of the trait attribute to be displayed is often enough to produce a usable result.

The following table lists the attributes of the Item class, organized by functional categories. Refer to the *Traits API Reference* for details on the Item class.

Attributes of Item, by category

Category	Attributes	Description
Content	• **name**	These attributes specify the actual data to be displayed by an item. Because an Item is essentially a template for displaying a single trait, its **name** attribute is nearly always specified.
Display format	• **dock** • **emphasized** • **export** • **height** • **image** • **item_theme** • **label** • **label_theme** • **padding** • **resizable** • **show_label** • **springy** • **width**	In addition to specifying which trait attributes are to be displayed, you might need to adjust the format of one or more of the resulting widgets. If an Item's **label** attribute is specified but not its name, the value of **label** is displayed as a simple, non-editable string. (This feature can be useful for displaying comments or instructions in a TraitsUI window.)
Content format	• **format_str** • **format_func**	In some cases it can be desirable to apply special formatting to a widget's contents rather than to the widget itself. Examples of such formatting might include rounding a floating-point value to two decimal places, or capitalizing all letter characters in a license plate number.
Widget override	• **editor** • **style**	These attributes override the widget that is automatically selected by TraitsUI. These options are discussed in *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories*.
Visibility and status	• **enabled_when** • **visible_when** • **defined_when** • **has_focus**	Use these attributes to create a simple form of a dynamic GUI, which alters the display in response to changes in the data it contains. More sophisticated dynamic behavior can be implemented using a custom *Handler* see *Controlling the Interface: the Handler*).
User help	• **tooltip** • **help** • **help_id**	These attributes provide guidance to the user in using the user interface. If the **help** attribute is not defined for an Item, a system-generated message is used instead. The **help_id** attribute is ignored by the default help handler, but can be used by a custom help handler.
Unique identifier	• **id**	The **id** attribute is used as a key for saving user preferences about the widget. If **id** is not specified, the value of the **name** attribute is used.

Subclasses of Item

The TraitsUI package defines the following subclasses of Item:

- Label
- Heading
- Spring

These classes are intended to help with the layout of a TraitsUI View, and need not have a trait attribute associated with them. See the *Traits API Reference* for details.

The Group Object

The preceding sections have shown how to construct windows that display a simple vertical sequence of widgets using instances of the View and Item classes. For more sophisticated interfaces, though, it is often desirable to treat a group of data elements as a unit for reasons that might be visual (e.g., placing the widgets within a labeled border) or logical (activating or deactivating the widgets in response to a single condition, defining group-level help text). In TraitsUI, such grouping is accomplished by means of the *Group* object.

Consider the following enhancement to Example 2:

pair: configure_traits(); examples triple: View; Group; examples

Example 3: Using configure_traits() with a View and a Group object

```
# configure_traits_view_group.py -- Sample code to demonstrate
#                                   configure_traits()
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

view1 = View(Group(Item(name = 'first_name'),
                   Item(name = 'last_name'),
                   Item(name = 'department'),
                   label = 'Personnel profile',
                   show_border = True))

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting window shows the same widgets as before, but they are now enclosed in a visible border with a text label:

Fig. 1.3: Figure 3: User interface for Example 3

Content of a Group

The content of a Group object is specified exactly like that of a View object. In other words, one or more Item or Group objects are given as arguments to the Group constructor, e.g., the three Items in Example 3. [5] The objects contained in a Group are called the *elements* of that Group. Groups can be nested to any level.

Group Attributes

The following table lists the attributes of the Group class, organized by functional categories. As with Item attributes, many of these attributes can be left unspecified for any given Group, as the default values usually lead to acceptable displays and behavior.

See the *Traits API Reference* for details of the Group class.

[5] As with Views, it is possible for a Group to contain objects of more than one type, but it is not recommended.

Attributes of Group, by category

Category	Attributes	Description
Content	• **object** • **content**	The **object** attribute references the object whose traits are being edited by members of the group; by default this is 'object', but could be another object in the current context. The **content** attribute is a list of elements in the group.
Display format	• **columns** • **dock** • **dock_theme** • **export** • **group_theme** • **image** • **item_theme** • **label** • **label_theme** • **layout** • **orientation** • **padding** • **selected** • **show_border** • **show_labels** • **show_left** • **springy** • **style**	These attributes define display options for the group as a whole.
Visibility and status	• **enabled_when** • **visible_when** • **defined_when**	These attributes work similarly to the attributes of the same names on the Item class.

User help	• **help**
	• **help_id**
		The help text is used by the default help handler only if the group is the only top-level group for the current View. For example, suppose help text is defined for a Group called **group1**. The following View shows this text in its help window:
		`View(group1)`
		The following two do not:
		`View(group1, group2)`
		`View(Group(group1))`
		The **help_id** attribute is ignored by the default help handler, but can be used by a custom help handler.

Unique identifier	• **id**	The **id** attribute is used as a key for saving user preferences about the widget. If **id** is not specified, the **id** values of the elements of the group are concatenated and used as the group identifier.

Subclasses of Group

The TraitsUI package defines the following subclasses of Group, which are helpful shorthands for defining certain types of groups. Refer to the *Traits API Reference* for details.

Subclasses of Group

Sub-class	Description	Equivalent To
HGroup	A group whose items are laid out horizontally.	`Group(orientation='horizontal')`
HFlow	A horizontal group whose items "wrap" when they exceed the available horizontal space.	`Group(orientation='horizontal', layout='flow', show_labels=False)`
HSplit	A horizontal group with splitter bars to separate it from other groups.	`Group(orientation='horizontal', layout='split')`
Tabbed	A group that is shown as a tab in a notebook.	`Group(orientation='horizontal' layout='tabbed', springy=True)`
VGroup	A group whose items are laid out vertically.	`Group(orientation='vertical')`
VFlow	A vertical group whose items "wrap" when they exceed the available vertical space.	`Group(orientation='vertical', layout='flow', show_labels=False)`
VFold	A vertical group in which items can be collapsed (i.e., folded) by clicking their titles.	`Group(orientation='vertical', layout='fold', show_labels=False)`
VGrid	A vertical group whose items are laid out in two columns.	`Group(orientation='vertical', columns=2)`
VS-plit	A vertical group with splitter bars to separate it from other groups.	`Group(orientation='vertical', layout='split')`

1.4 Customizing a View

As shown in the preceding two chapters, it is possible to specify a window in TraitsUI simply by creating a View object with the appropriate contents. In designing real-life applications, however, you usually need to be able to control the appearance and behavior of the windows themselves, not merely their content. This chapter covers a variety of options for tailoring the appearance of a window that is created using a View, including the type of window that a View appears in, the *command button*s that appear in the window, and the physical properties of the window.

1.4.1 Specifying Window Type: the kind Attribute

Many types of windows can be used to display the same data content. A form can appear in a window, a wizard, or an embedded panel; windows can be *modal* (i.e., stop all other program processing until the box is dismissed) or not, and can interact with live data or with a buffered copy. In TraitsUI, a single View can be used to implement any of these options simply by modifying its **kind** attribute. There are seven possible values of **kind**:

- 'modal'
- 'live'
- 'livemodal'
- 'nonmodal'
- 'wizard'
- 'panel'
- 'subpanel'

These alternatives are described below. If the **kind** attribute of a View object is not specified, the default value is 'modal'.

Stand-alone Windows

The behavior of a stand-alone TraitsUI window can vary over two significant degrees of freedom. First, it can be *modal*, meaning that when the window appears, all other GUI interaction is suspended until the window is closed; if it is not modal, then both the window and the rest of the GUI remain active and responsive. Second, it can be *live*, meaning that any changes that the user makes to data in the window is applied directly and immediately to the underlying model object or objects; otherwise the changes are made to a copy of the model data, and are only copied to the model when the user commits them (usually by clicking an *OK* or *Apply* button; see *Command Buttons: the buttons Attribute*). The four possible combinations of these behaviors correspond to four of the possible values of the 'kind ' attribute of the View object, as shown in the following table.

Matrix of TraitsUI windows

	not modal	modal
not live	*nonmodal*	*modal*
live	*live*	*livemodal*

All of these window types are identical in appearance. Also, all types support the **buttons** attribute, which is described in *Command Buttons: the buttons Attribute*. Usually, a window with command buttons is called a *dialog box*.

Wizards

Unlike a window, whose contents generally appear as a single page or a tabbed display, a *wizard* is presented as a series of pages that a user must navigate sequentially.

TraitsUI Wizards are always modal and live. They always display a standard wizard button set; i.e., they ignore the **buttons** View attribute. In short, wizards are considerably less flexible than windows, and are primarily suitable for highly controlled user interactions such as software installation.

Panels and Subpanels

Both dialog boxes and wizards are secondary windows that appear separately from the main program display, if any. Often, however, you might need to create a window element that is embedded in a larger display. For such cases, the **kind** of the corresponding View object should be 'panel' or 'subpanel '.

A *panel* is very similar to a window, except that it is embedded in a larger window, which need not be a TraitsUI window. Like windows, panels support the **buttons** View attribute, as well as any menus and toolbars that are specified for the View (see *Menus and Menu Bars*). Panels are always live and nonmodal.

A *subpanel* is almost identical to a panel. The only difference is that subpanels do not display *command button*s even if the View specifies them.

1.4.2 Command Buttons: the buttons Attribute

A common feature of many windows is a row of command buttons along the bottom of the frame. These buttons have a fixed position outside any scrolled panels in the window, and are thus always visible while the window is displayed. They are usually used for window-level commands such as committing or cancelling the changes made to the form data, or displaying a help window.

In TraitsUI, these command buttons are specified by means of the View object's **buttons** attribute, whose value is a list of buttons to display. [6] Consider the following variation on Example 3:

Example 4: Using a View object with buttons

```
# configure_traits_view_buttons.py -- Sample code to demonstrate
#                                     configure_traits()

from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item
from traitsui.menu import OKButton, CancelButton

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

view1 = View(Item(name = 'first_name'),
             Item(name = 'last_name'),
             Item(name = 'department'),
             buttons = [OKButton, CancelButton])

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting window has the same content as before, but now two buttons are displayed at the bottom: *OK* and *Cancel*:

Fig. 1.4: Figure 4: User interface for Example 4

There are six standard buttons defined by TraitsUI. Each of the standard buttons has matching a string alias. You can either import and use the button names, or simply use their aliases:

[6] Actually, the value of the **buttons** attribute is really a list of Action objects, from which GUI buttons are generated by TraitsUI. The Action class is described in *Actions*.

Command button aliases

Button Name	Button Alias
UndoButton	'Undo'
ApplyButton	'Apply'
RevertButton	'Revert'
OKButton	'OK' (case sensitive!)
CancelButton	'Cancel'

Alternatively, there are several pre-defined button lists that can be imported from traitsui.menu and assigned to the buttons attribute:

- OKCancelButtons = `[OKButton, CancelButton ]`
- ModalButtons = `[ ApplyButton, RevertButton, OKButton, CancelButton, HelpButton ]`
- LiveButtons = `[ UndoButton, RevertButton, OKButton, CancelButton, HelpButton ]`

Thus, one could rewrite the lines in Example 4 as follows, and the effect would be exactly the same:

```
from traitsui.menu import OKCancelButtons

            buttons = OKCancelButtons
```

The special constant NoButtons can be used to create a window or panel without command buttons. While this is the default behavior, NoButtons can be useful for overriding an explicit value for **buttons**. You can also specify `buttons = []` to achieve the same effect. Setting the **buttons** attribute to an empty list has the same effect as not defining it at all.

It is also possible to define custom buttons and add them to the **buttons** list; see *Custom Command Buttons* for details.

1.4.3 Other View Attributes

Attributes of View, by category

Category	Attributes	Description
Window display	• **dock** • **height** • **icon** • **image** • **item_theme** • **label_theme** • **resizable** • **scrollable** • **statusbar** • **style** • **title** • **width** • **x** • **y**	These attributes control the visual properties of the window itself, regardless of its content.
Command	• **close_result** • **handler** • **key_bindings** • **menubar** • **model_view** • **on_apply** • **toolbar** • **updated**	TraitsUI menus and toolbars are generally implemented in conjunction with custom *Handler*s; see *Menus and Menu Bars* for details. The **key_bindings** attribute references the set of global key bindings for the view.
Content	• **content** • **drop_class** • **export** • **imports** • **object**	The **content** attribute is the top-level Group object for the view. The **object** attribute is the object being edited. The **imports** and **drop_class** attributes control what objects can be dragged and dropped on the view.
User help	• **help** • **help_id**	The **help** attribute is a deprecated way to specify that the View has a Help button. Use the buttons attribute instead (see *Command Buttons: the buttons Attribute* for details). The **help_id** attribute is not used by Traits, but can be used by a custom help handler.
Unique identifier	• **id**	The **id** attribute is used as a key to save user preferences about a view, such as customized size and position, so that they are restored the next time the view is opened. The value of **id** must be unique across all Traits-based applications on a system. If no value is specified, no user preferences are saved for the view.

1.5 Advanced View Concepts

The preceding chapters of this Manual give an overview of how to use the View class to quickly construct a simple window for a single HasTraits object. This chapter explores a number of more complex techniques that significantly increase the power and versatility of the View object.

- *Internal Views:* Views can be defined as attributes of a HasTraits class; one class can have multiple views. View attributes can be inherited by subclasses.
- *External Views:* A view can be defined as a module variable, inline as a function or method argument, or as an attribute of a *Handler*.
- *Ways of displaying Views:* You can display a View by calling configure_traits() or edit_traits() on a HasTraits object, or by calling the ui() method on the View object.
- *View context:* You can pass a context to any of the methods for displaying views, which is a dictionary of labels and objects. In the default case, this dictionary contains only one object, referenced as 'object', but you can define contexts that contain multiple objects.
- *Include objects:* You can use an Include object as a placeholder for view items defined elsewhere.

1.5.1 Internal Views

In the examples thus far, the View objects have been external. That is to say, they have been defined outside the model (HasTraits object or objects) that they are used to display. This approach is in keeping with the separation of the two concepts prescribed by the *MVC* design pattern.

There are cases in which it is useful to define a View within a HasTraits class. In particular, it can be useful to associate one or more Views with a particular type of object so that they can be incorporated into other parts of the application with little or no additional programming. Further, a View that is defined within a model class is inherited by any subclasses of that class, a phenomenon called *visual inheritance*.

Defining a Default View

It is easy to define a default view for a HasTraits class: simply create a View attribute called **traits_view** for that class. Consider the following variation on Example 3:

Example 5: Using configure_traits() with a default View object

```
# default_traits_view.py -- Sample code to demonstrate the use of
#                           'traits_view'
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

class SimpleEmployee2(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

    traits_view = View(Group(Item(name = 'first_name'),
                             Item(name = 'last_name'),
```

```
                                Item(name = 'department'),
                                label = 'Personnel profile',
                                show_border = True))

sam = SimpleEmployee2()
sam.configure_traits()
```

In this example, configure_traits() no longer requires a *view* keyword argument; the **traits_view** attribute is used by default, resulting in the same display as in Figure 3:

Fig. 1.5: Figure 5: User interface for Example 5

It is not strictly necessary to call this View attribute **traits_view**. If exactly one View attribute is defined for a HasTraits class, that View is always treated as the default display template for the class. However, if there are multiple View attributes for the class (as discussed in the next section), if one is named 'traits_view', it is always used as the default.

Sometimes, it is necessary to build a view based on the state of the object when it is being built. In such cases, defining the view statically is limiting, so one can override the default_traits_view() method of a HasTraits object. The example above would be implemented as follows:

Example 5b: Building a default View object with default_traits_view()

```
# default_traits_view2.py -- Sample code to demonstrate the use of
#                            'default_traits_view'
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

class SimpleEmployee2(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

    def default_traits_view(self):
        return View(Group(Item(name = 'first_name'),
                          Item(name = 'last_name'),
                          Item(name = 'department'),
                          label = 'Personnel profile',
                          show_border = True))

sam = SimpleEmployee2()
sam.configure_traits()
```

This pattern can be useful for situations where the layout of GUI elements depends on the state of the object. For instance, to populate the values of a *CheckListEditor()* with items read in from a file, it would be useful to build the default view this way.

Defining Multiple Views Within the Model

Sometimes it is useful to have more than one pre-defined view for a model class. In the case of the SimpleEmployee class, one might want to have both a "public information" view like the one above and an "all information" view. One can do this by simply adding a second View attribute:

Example 6: Defining multiple View objects in a HasTraits class

```
# multiple_views.py -- Sample code to demonstrate the use of
#                      multiple views
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

class SimpleEmployee3(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

    traits_view = View(Group(Item(name = 'first_name'),
                             Item(name = 'last_name'),
                             Item(name = 'department'),
                             label = 'Personnel profile',
                             show_border = True))

    all_view = View(Group(Item(name = 'first_name'),
                          Item(name = 'last_name'),
                          Item(name = 'department'),
                          Item(name = 'employee_number'),
                          Item(name = 'salary'),
                          label = 'Personnel database ' +
                                  'entry',
                          show_border = True))

sam = SimpleEmployee3()
sam.configure_traits()
sam.configure_traits(view='all_view')
```

As before, a simple call to configure_traits() for an object of this class produces a window based on the default View (**traits_view**). In order to use the alternate View, use the same syntax as for an external view, except that the View name is specified in single quotes to indicate that it is associated with the object rather than being a module-level variable:

```
configure_traits(view='all_view').
```

Note that if more than one View is defined for a model class, you must indicate which one is to be used as the default by naming it `traits_view`. Otherwise, TraitsUI gives preference to none of them, and instead tries to construct a default View, resulting in a simple alphabetized display as described in *The View and Its Building Blocks*. For this

reason, it is usually preferable to name a model's default View traits_view even if there are no other Views; otherwise, simply defining additional Views, even if they are never used, can unexpectedly change the behavior of the GUI.

1.5.2 Separating Model and View: External Views

In all the preceding examples in this guide, the concepts of model and view have remained closely coupled. In some cases the view has been defined in the model class, as in *Internal Views*; in other cases the configure_traits() method that produces a window from a View has been called from a HasTraits object. However, these strategies are simply conveniences; they are not an intrinsic part of the relationship between model and view in TraitsUI. This section begins to explore how the TraitsUI package truly supports the separation of model and view prescribed by the *MVC* design pattern.

An *external* view is one that is defined outside the model classes. In Traits UI, you can define a named View wherever you can define a variable or class attribute. [7] A View can even be defined in-line as a function or method argument, for example:

```
object.configure_traits(view=View(Group(Item(name='a'),
                                        Item(name='b'),
                                        Item(name='c'))))
```

However, this approach is apt to obfuscate the code unless the View is very simple.

Example 2 through *Example 4* demonstrate external Views defined as variables. One advantage of this convention is that the variable name provides an easily accessible "handle" for re-using the View. This technique does not, however, support visual inheritance.

A powerful alternative is to define a View within the *controller* (Handler) class that controls the window for that View. [8] This technique is described in *Controlling the Interface: the Handler*.

1.5.3 Displaying a View

TraitsUI provides three methods for creating a window or panel from a View object. The first two, configure_traits() and edit_traits(), are defined on the HasTraits class, which is a superclass of all Traits-based model classes, as well as of Handler and its subclasses. The third method, ui(), is defined on the View class itself.

configure_traits()

The configure_traits() method creates a standalone window for a given View object, i.e., it does not require an existing GUI to run in. It is therefore suitable for building command-line functions, as well as providing an accessible tool for the beginning TraitsUI programmer.

The configure_traits() method also provides options for saving *trait attribute* values to and restoring them from a file. Refer to the *Traits API Reference* for details.

edit_traits()

The edit_traits() method is very similar to configure_traits(), with two major exceptions. First, it is designed to run from within a larger application whose GUI is already defined. Second, it does not provide options for saving data to and restoring data from a file, as it is assumed that these operations are handled elsewhere in the application.

[7] Note that although the definition of a View within a HasTraits class has the syntax of a trait attribute definition, the resulting View is not stored as an attribute of the class.

[8] Assuming there is one; not all GUIs require an explicitly defined Handler.

ui()

The View object includes a method called ui(), which performs the actual generation of the window or panel from the View for both edit_traits() and configure_traits(). The ui() method is also available directly through the TraitsUI API; however, using one of the other two methods is usually preferable. [9]

The ui() method has five keyword parameters:

- *kind*
- *context*
- *handler*
- *parent*
- *view_elements*

The first four are identical in form and function to the corresponding arguments of edit_traits(), except that *context* is not optional; the following section explains why.

The fifth argument, *view_elements*, is used only in the context of a call to ui() from a model object method, i.e., from configure_traits() or edit_traits(), Therefore it is irrelevant in the rare cases when ui() is used directly by client code. It contains a dictionary of the named *ViewElement* objects defined for the object whose configure_traits() (or edit_traits()) method was called..

1.5.4 The View Context

All three of the methods described in *Displaying a View* have a *context* parameter. This parameter can be a single object or a dictionary of string/object pairs; the object or objects are the model objects whose traits attributes are to be edited. In general a "context" is a Python dictionary whose keys are strings; the key strings are used to look up the values. In the case of the *context* parameter to the ui() method, the dictionary values are objects. In the special case where only one object is relevant, it can be passed directly instead of wrapping it in a dictionary.

When the ui() method is called from configure_traits() or edit_traits() on a HasTraits object, the relevant object is the HasTraits object whose method was called. For this reason, you do not need to specify the *context* argument in most calls to configure_traits() or edit_traits(). However, when you call the ui() method on a View object, you *must* specify the *context* parameter, so that the ui() method receives references to the objects whose trait attributes you want to modify.

So, if configure_traits() figures out the relevant context for you, why call ui() at all? One answer lies in *multi-object Views*.

Multi-Object Views

A multi-object view is any view whose contents depend on multiple "independent" model objects, i.e., objects that are not attributes of one another. For example, suppose you are building a real estate listing application, and want to display a window that shows two properties side by side for a comparison of price and features. This is straightforward in TraitsUI, as the following example shows:

Example 7: Using a multi-object view with a context

[9] One possible exception is the case where a View object is defined as a variable (i.e., outside any class) or within a custom Handler, and is associated more or less equally with multiple model objects; see *Multi-Object Views*.

```
# multi_object_view.py -- Sample code to show multi-object view
#                         with context

from traits.api import HasTraits, Str, Int, Bool
from traitsui.api import View, Group, Item

# Sample class
class House(HasTraits):
   address = Str
   bedrooms = Int
   pool = Bool
   price = Int

# View object designed to display two objects of class 'House'
comp_view = View(
    Group(
        Group(
            Item('h1.address', resizable=True),
            Item('h1.bedrooms'),
            Item('h1.pool'),
            Item('h1.price'),
            show_border=True
        ),
        Group(
            Item('h2.address', resizable=True),
            Item('h2.bedrooms'),
            Item('h2.pool'),
            Item('h2.price'),
            show_border=True
        ),
        orientation = 'horizontal'
    ),
    title = 'House Comparison'
)
# A pair of houses to demonstrate the View
house1 = House(address='4743 Dudley Lane',
               bedrooms=3,
               pool=False,
               price=150000)
house2 = House(address='11604 Autumn Ridge',
               bedrooms=3,
               pool=True,
               price=200000)

# ...And the actual display command
house1.configure_traits(view=comp_view, context={'h1':house1,
                                                 'h2':house2})
```

The resulting window has the desired appearance: [10]

For the purposes of this particular example, it makes sense to create a separate Group for each model object, and to use two model objects of the same class. Note, however, that neither is a requirement.

Notice that the Item definitions in Example 7 use the same type of extended trait attribute syntax as is supported for the on_trait_change() dynamic trait change notification method. In fact, Item **name** attributes can reference any trait

[10] If the script were designed to run within an existing GUI, it would make sense to replace the last line with `comp_view.ui(context={'h1': house1, 'h2': house2})`, since neither object particularly dominates the view. However, the examples in this Manual are designed to be fully executable from the Python command line, which is why configure_traits() was used instead.

Fig. 1.6: Figure 6: User interface for Example 7

attribute that is reachable from an object in the context. This is true regardless of whether the context contains a single object or multiple objects. For example:

```
Item('object.axle.chassis.serial_number')
```

where *"object"* is the literal name which refers to the top-level object being viewed. (Note that *"object"* is **not** some user-defined attribute name like *"axle"* in this example.) More precisely, *"object"* is the default name, in the view's *context* dictionary, of this top-level viewed object (see *Advanced View Concepts*).

Because an Item can refer only to a single trait, do not use extended trait references that refer to multiple traits, since the behavior of such references is not defined. Also, avoid extended trait references where one of the intermediate objects could be None, because there is no way to obtain a valid reference from None.

Refer to the Traits User Manual, in the chapter on trait notification, for details of the extended trait name syntax.

1.5.5 Include Objects

In addition to the Item and Group class, a third building block class for Views exists in TraitsUI: the Include class. For the sake of completeness, this section gives a brief description of Include objects and their purpose and usage. However, they are not commonly used as of this writing, and should be considered unsupported pending redesign.

In essence, an Include object is a placeholder for a named Group or Item object that is specified outside the Group or View in which it appears. For example, the following two definitions, taken together, are equivalent to the third:

Example 8: Using an Include object

```
# This fragment...
my_view = View(Group(Item('a'),
                     Item('b')),
               Include('my_group'))

# ...plus this fragment...
my_group = Group(Item('c'),
                 Item('d'),
                 Item('e'))

#...are equivalent to this:
my_view = View(Group(Item('a'),
                     Item('b')),
               Group(Item('c'),
                     Item('d'),
                     Item('e'))
```

This opens an interesting possibility when a View is part of a model class: any Include objects belonging to that View can be defined differently for different instances or subclasses of that class. This technique is called *view parameterization*.

1.6 Controlling the Interface: the Handler

Most of the material in the preceding chapters is concerned with the relationship between the model and view aspects of the *MVC* design pattern as supported by TraitsUI. This chapter examines the third aspect: the *controller*, implemented in TraitsUI as an *instance* of the *Handler* class. [11]

A controller for an MVC-based application is essentially an event handler for GUI events, i.e., for events that are generated through or by the program interface. Such events can require changes to one or more model objects (e.g., because a data value has been updated) or manipulation of the interface itself (e.g., window closure, dynamic interface behavior). In TraitsUI, such actions are performed by a Handler object.

In the preceding examples in this guide, the Handler object has been implicit: TraitsUI provides a default Handler that takes care of a common set of GUI events including window initialization and closure, data value updates, and button press events for the standard TraitsUI window buttons (see *Command Buttons: the buttons Attribute*).

This chapter explains the features of the TraitsUI Handler, and shows how to implement custom GUI behaviors by building and instantiating custom subclasses of the Handler class. The final section of the chapter describes several techniques for linking a custom Handler to the window or windows it is designed to control.

1.6.1 Backstage: Introducing the UIInfo Object

TraitsUI supports the MVC design pattern by maintaining the model, view, and controller as separate entities. A single View object can be used to construct windows for multiple model objects; likewise a single Handler can handle GUI events for windows created using different Views. Thus there is no static link between a Handler and any particular window or model object. However, in order to be useful, a Handler must be able to observe and manipulate both its corresponding window and model objects. In TraitsUI, this is accomplished by means of the UIInfo object.

Whenever TraitsUI creates a window or panel from a View, a UIInfo object is created to act as the Handler's reference to that window and to the objects whose *trait attribute*s are displayed in it. Each entry in the View's context (see *The View Context*) becomes an attribute of the UIInfo object. [12] For example, the UIInfo object created in *Example 7* has attributes **h1** and **h2** whose values are the objects **house1** and **house2** respectively. In *Example 1* through *Example 6*, the created UIInfo object has an attribute **object** whose value is the object **sam**.

Whenever a window event causes a Handler method to be called, TraitsUI passes the corresponding UIInfo object as one of the method arguments. This gives the Handler the information necessary to perform its tasks.

1.6.2 Assigning Handlers to Views

In accordance with the MVC design pattern, Handlers and Views are separate entities belonging to distinct classes. In order for a custom Handler to provide the control logic for a window, it must be explicitly associated with the View for that window. The TraitsUI package provides three ways to accomplish this:

- Make the Handler an attribute of the View.
- Provide the Handler as an argument to a display method such as edit_traits().
- Define the View as part of the Handler.

[11] Except those implemented via the **enabled_when**, **visible_when**, and **defined_when** attributes of Items and Groups.

[12] Other attributes of the UIInfo object include a UI object and any *trait editors* contained in the window (see *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories*).

Binding a Singleton Handler to a View

To associate a given custom Handler with all windows produced from a given View, assign an instance of the custom Handler class to the View's **handler** attribute. The result of this technique, as shown in *Example 9*, is that the window created by the View object is automatically controlled by the specified handler instance.

Linking Handler and View at Edit Time

It is also possible to associate a custom Handler with a specific window without assigning it permanently to the View. Each of the three TraitsUI window-building methods (the configure_traits() and edit_traits() methods of the HasTraits class and the ui() method of the View class) has a *handler* keyword argument. Assigning an instance of Handler to this argument gives that handler instance control *only of the specific window being created by the method call*. This assignment overrides the View's **handler** attribute.

Creating a Default View Within a Handler

You seldom need to associate a single custom Handler with several different Views or vice versa, although you can in theory and there are cases where it is useful to be able to do so. In most real-life scenarios, a custom Handler is tailored to a particular View with which it is always used. One way to reflect this usage in the program design is to define the View as part of the Handler. The same rules apply as for defining Views within HasTraits objects; for example, a view named 'trait_view' is used as the default view.

The Handler class, which is a subclass of HasTraits, overrides the standard configure_traits() and edit_traits() methods; the subclass versions are identical to the originals except that the Handler object on which they are called becomes the default Handler for the resulting windows. Note that for these versions of the display methods, the *context* keyword parameter is not optional.

1.6.3 Handler Subclasses

TraitsUI provides two Handler subclasses: ModelView and Controller. Both of these classes are designed to simplify the process of creating an MVC-based application.

Both ModelView and Controller extend the Handler class by adding the following trait attributes:

- **model**: The model object for which this handler defines a view and controller.
- **info**: The UIInfo object associated with the actual user interface window or panel for the model object.

The **model** attribute provides convenient access to the model object associated with either subclass. Normally, the **model** attribute is set in the constructor when an instance of ModelView or Controller is created.

The **info** attribute provides convenient access to the UIInfo object associated with the active user interface view for the handler object. The **info** attribute is automatically set when the handler object's view is created.

Both classes' constructors accept an optional *model* parameter, which is the model object. They also can accept metadata as keyword parameters.

class `ModelView` ([*model* = *None*, ***metadata*])

class `Controller` ([*model* = *None*, ***metadata*])

The difference between the ModelView and Controller classes lies in the context dictionary that each one passes to its associated user interface, as described in the following sections.

Controller Class

The Controller class is normally used when implementing a standard MVC-based design, and plays the "controller" role in the MVC design pattern. The "model" role is played by the object referenced by the Controller's **model** attribute; and the "view" role is played by the View object associated with the model object.

The context dictionary that a Controller object passes to the View's ui() method contains the following entries:

- `object`: The Controller's model object.
- `controller`: The Controller object itself.

Using a Controller as the handler class assumes that the model object contains most, if not all, of the data to be viewed. Therefore, the model object is used for the object key in the context dictionary, so that its attributes can be easily referenced with unqualified names (such as Item('name')).

ModelView Class

The ModelView class is useful when creating a variant of the standard MVC design pattern. In this variant, the ModelView subclass reformulates a number of trait attributes on its model object as properties on the ModelView, usually to convert the model's data into a format that is more suited to a user interface.

The context dictionary that a ModelView object passes to the View's ui() method contains the following entries:

- `object`: The ModelView object itself.
- `model`: The ModelView's model object.

In effect, the ModelView object substitutes itself for the model object in relation to the View object, serving both the "controller" role and the "model" role (as a set of properties wrapped around the original model). Because the ModelView object is passed as the context's object, its attributes can be referenced by unqualified names in the View definition.

1.6.4 Writing Handler Methods

If you create a custom Handler subclass, depending on the behavior you want to implement, you might override the standard methods of Handler, or you might create methods that respond to changes to specific trait attributes.

Overriding Standard Methods

The Handler class provides methods that are automatically executed at certain points in the lifespan of the window controlled by a given Handler. By overriding these methods, you can implement a variety of custom window behaviors. The following sequence shows the points at which the Handler methods are called.

1. A UIInfo object is created
2. The Handler's init_info() method is called. Override this method if the handler needs access to viewable traits on the UIInfo object whose values are properties that depend on items in the context being edited.
3. The UI object is created, and generates the actual window.
4. The init() method is called. Override this method if you need to initialize or customize the window.
5. The position() method is called. Override this method to modify the position of the window (if setting the x and y attributes of the View is insufficient).
6. The window is displayed.

When Handler methods are called, and when to override them

Method	Called When	Override When?
apply(info)	The user clicks the *Apply* button, and after the changes have been applied to the context objects.	To perform additional processing at this point.
close(info, is_ok)	The user requests to close the window, clicking *OK*, *Cancel*, or the window close button, menu, or icon.	To perform additional checks before destroying the window.
closed(info, is_ok)	The window has been destroyed.	To perform additional clean-up tasks.
revert(info)	The user clicks the *Revert* button, or clicks *Cancel* in a live window.	To perform additional processing.
setattr(info, object, name, value)	The user changes a trait attribute value through the user interface.	To perform additional processing, such as keeping a change history. Make sure that the overriding method actually sets the attribute.
show_help(info, control=None)	The user clicks the *Help* button.	To call a custom help handler in addition to or instead of the global help handler, for this window.

Reacting to Trait Changes

The setattr() method described above is called whenever any trait value is changed in the UI. However, TraitsUI also provides a mechanism for calling methods that are automatically executed whenever the user edits a *particular* trait. While you can use static notification handler methods on the HasTraits object, you might want to implement behavior that concerns only the user interface. In that case, following the MVC pattern dictates that such behavior should not be implemented in the "model" part of the code. In keeping with this pattern, TraitsUI supports "user interface notification" methods, which must have a signature with the following format:

`extended_traitname_changed`(*info*)

This method is called whenever a change is made to the attribute specified by *extended_traitname* in the **context** of the View used to create the window (see *Multi-Object Views*), where the dots in the extended trait reference have been replaced by underscores. For example, for a method to handle changes on the **salary** attribute of the object whose context key is 'object' (the default object), the method name should be object_salary_changed().

By contrast, a subclass of Handler for *Example 7* might include a method called h2_price_changed() to be called whenever the price of the second house is edited.

Note: These methods are called on window creation.

User interface notification methods are called when the window is first created.

To differentiate between code that should be executed when the window is first initialized and code that should be executed when the trait actually changes, use the **initialized** attribute of the UIInfo object (i.e., of the *info* argument):

```
def object_foo_changed(self, info):

    if not info.initialized:
        #code to be executed only when the window is
        #created
    else:
        #code to be executed only when 'foo' changes after
        #window initialization}
```

```
    #code to be executed in either case
```

The following script, which annotates its window's title with an asterisk ('*') the first time a data element is updated, demonstrates a simple use of both an overridden setattr() method and user interface notification method.

Example 9: Using a Handler that reacts to trait changes

```
# handler_override.py -- Example of a Handler that overrides
#                        setattr(), and that has a user interface
#                        notification method

from traits.api import HasTraits, Bool
from traitsui.api import View, Handler

class TC_Handler(Handler):

    def setattr(self, info, object, name, value):
        Handler.setattr(self, info, object, name, value)
        info.object._updated = True

    def object__updated_changed(self, info):
        if info.initialized:
            info.ui.title += "*"

class TestClass(HasTraits):
    b1 = Bool
    b2 = Bool
    b3 = Bool
    _updated = Bool(False)

view1 = View('b1', 'b2', 'b3',
             title="Alter Title",
             handler=TC_Handler(),
             buttons = ['OK', 'Cancel'])

tc = TestClass()
tc.configure_traits(view=view1)
```

Implementing Custom Window Commands

Another use of a Handler is to define custom window actions, which can be presented as buttons, menu items, or toolbar buttons.

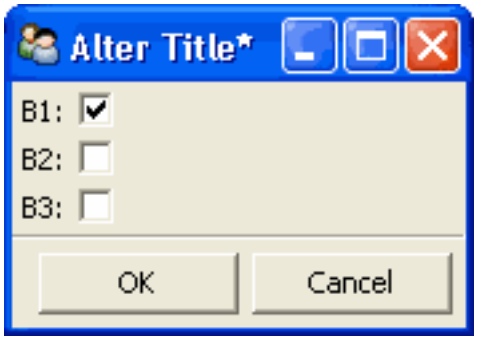


Fig. 1.7: Figure 7: Before and after views of Example 9

Actions

In TraitsUI, window commands are implemented as instances of the Action class. Actions can be used in *command button*s, menus, and toolbars.

Suppose you want to build a window with a custom **Recalculate** action. Suppose further that you have defined a subclass of Handler called MyHandler to provide the logic for the window. To create the action:

1. Add a method to MyHandler that implements the command logic. This method can have any name (e.g., do_recalc()), but must accept exactly one argument: a UIInfo object.
2. Create an Action instance using the name of the new method, e.g.:

```
recalc = Action(name = "Recalculate",
                action = "do_recalc")
```

Custom Command Buttons

The simplest way to turn an Action into a window command is to add it to the **buttons** attribute for the View. It appears in the button area of the window, along with any standard buttons you specify.

1. Define the handler method and action, as described in *Actions*.
2. Include the new Action in the **buttons** attribute for the View:

```
View ( #view contents,
       # ...,
       buttons = [ OKButton, CancelButton, recalc ])
```

Menus and Menu Bars

Another way to install an Action such as **recalc** as a window command is to make it into a menu option.

1. Define the handler method and action, as described in *Actions*.
2. If the View does not already include a MenuBar, create one and assign it to the View's **menubar** attribute.
3. If the appropriate Menu does not yet exist, create it and add it to the MenuBar.
4. Add the Action to the Menu.

These steps can be executed all at once when the View is created, as in the following code:

```
View ( #view contents,
       # ...,
       menubar = MenuBar(
          Menu( my_action,
                name = 'My Special Menu')))
```

Toolbars

A third way to add an action to a Traits View is to make it a button on a toolbar. Adding a toolbar to a Traits View is similar to adding a menu bar, except that toolbars do not contain menus; they directly contain actions.

1. Define the handler method and the action, as in *Actions*, including a tooltip and an image to display on the toolbar. The image must be a Pyface ImageResource instance; if a path to the image file is not specified, it is assumed to be in an images subdirectory of the directory where ImageResource is used:

```
From pyface.api import ImageResource

recalc = Action(name = "Recalculate",
                action = "do_recalc",
                toolip = "Recalculate the results",
                image = ImageResource("recalc.png"))
```

2. If the View does not already include a ToolBar, create one and assign it to the View's **toolbar** attribute.
3. Add the Action to the ToolBar.

As with a MenuBar, these steps can be executed all at once when the View is created, as in the following code:

```
View ( #view contents,
       # ...,
       toolbar = ToolBar( my_action))
```

1.7 TraitsUI Themes

Beginning in Traits 3.0, TraitsUI supports using *themes* to customize the appearance of user interfaces, by applying graphical elements extracted from simple images. For example, Figure 8 shows an unthemed Traits user interface.

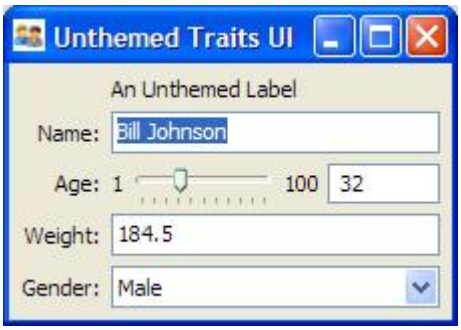


Fig. 1.8: Figure 8: Unthemed Traits user interface

Figure 9 shows the same user interface with a theme applied to it.

Figure 10 shows the same user interface with a different theme applied.

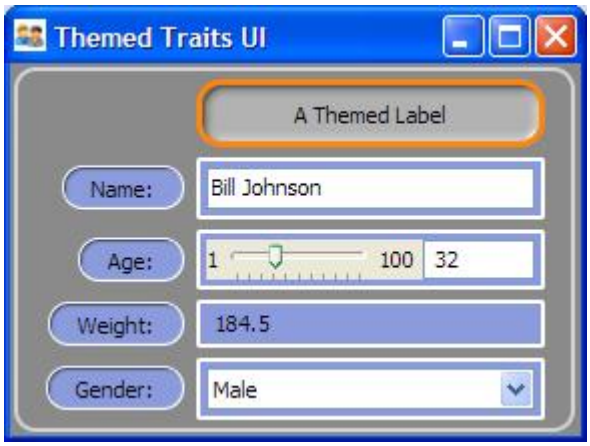


Fig. 1.9: Figure 9: Themed Traits user interface

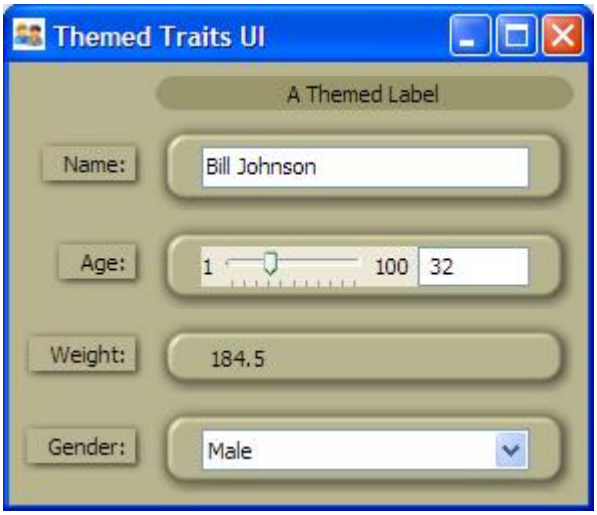


Fig. 1.10: Figure 10: Theme Traits user interface with alternate theme

1.7.1 Theme Data

All of the data used by TraitsUI for themes is in the form of simple images, a few examples of which are shown in Figure 11:

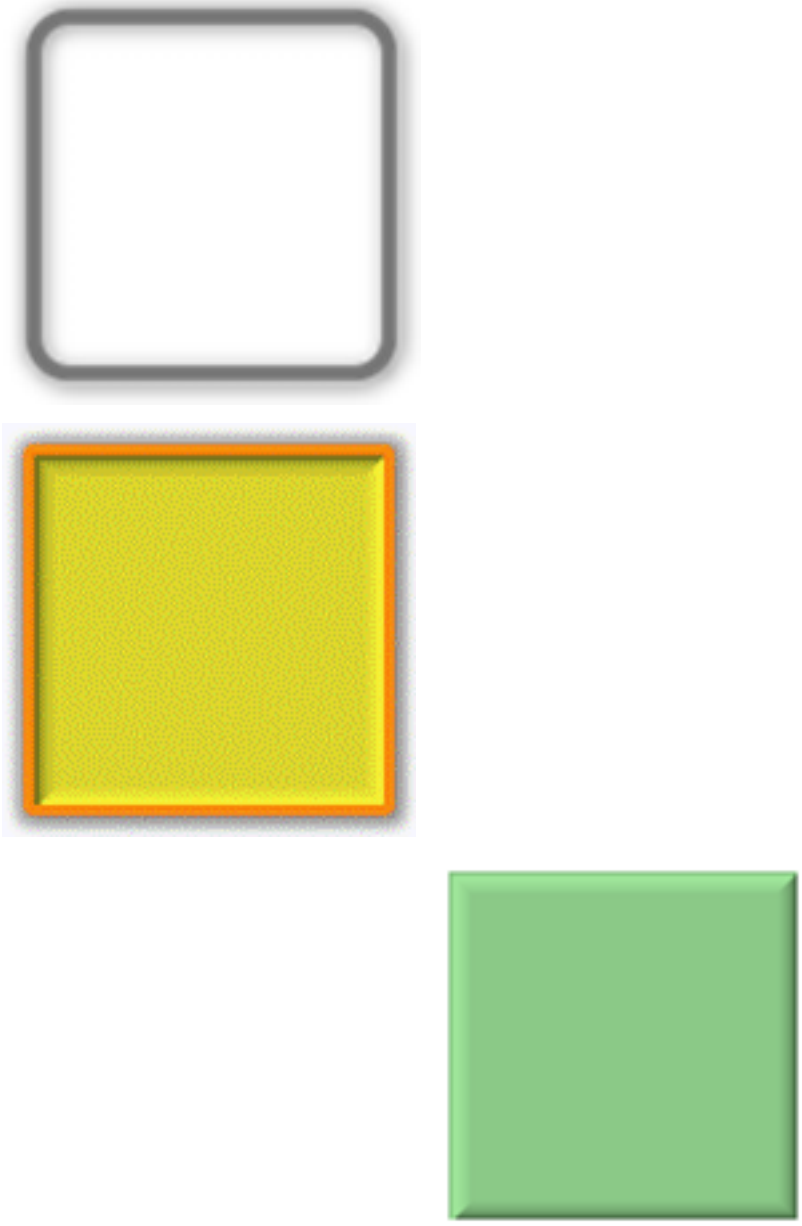


Fig. 1.11: Figure 11: Theme images

Any type of JPEG or Portable Network Graphics (PNG) file is supported. In particular, PNG files with alpha information allow smooth compositing of multiple theme images. The first image in Figure 11 is an example of a PNG file containing alpha information. That is, the interior of the rectangle is not gray, but transparent, with a thin alpha gradient shadow around its edges.

1.7.2 Themeable TraitsUI Elements

Theme information can be applied to the following classes of TraitsUI objects:

- *Group*
- *Item*
- *View*

All of these classes have **item_theme** and **label_theme** attributes, which specify the themes for an editor and its label, respectively; the Group class also has a **group_theme** attribute, which specifies the theme for the group itself. These attributes are defined to be Theme traits, which accept values which are either PyFace ImageResource objects, or strings that specify an image file to use. In the case of string values, no path information need be included. The path to the image file is assumed to be the images subdirectory or `images.zip` file located in the same directory as the source file containing the string. [13] However, if the string begins with an '@' (at-sign), the string is assumed to be a reference to an image in the default image library provided with PyFace. [14]

The **item_theme** and **label_theme** attributes are transferred via containment. That is, if an Item object has an **item_theme** defined, that value is used for the Item object's editor. If **item_theme** is not defined on the Item object, the **item_theme** value from the containing Group is used, and so on up to the **item_theme** value on containing View, if necessary. Therefore, it is possible to set the item and label themes for a whole user interface at the view level.

The **group_theme** attribute value is not transferred through containment, but nested groups automatically visually inherit the theme of the containing group. You can, of course, explicitly specify theme information at each level of a nested group hierarchy.

1.7.3 Adding Themes to a UI

To add themes to a Traits user interface, you add the theme-related attributes to the View, Group, and Item definitions. Example 10 shows the code for the unthemed user interface shown in Figure 8.

Example 10: TraitsUI without themes

```
# unthemed.py -- Example of a TraitsUI without themes
from traits.api import HasTraits, Str, Range, Float, Enum
from traitsui.api import View, Group, Item, Label
class Test ( HasTraits ):

    name   = Str
    age    = Range( 1, 100 )
    weight = Float
    gender = Enum( 'Male', 'Female' )

    view = View(
        Group(
            Label( 'An Unthemed Label' ),
            Item( 'name' ),
            Item( 'age' ),
            Item( 'weight' ),
            Item( 'gender' )
        ),
        title   = 'Unthemed TraitsUI',
```

[13] This is very similar to the way that PyFace ImageResource objects work when no search path is specified.

[14] PyFace is provided by the pyface package in the Traits GUI project (not to be confused with the TraitsUI package, traitsui, the subject of this document.)

```
    )

Test().configure_traits()
```

Example 11 shows the code for the user interface shown in Figure 9, which is essentially the same as in Example 10, but with theme data added.

Example 11: TraitsUI with themes

```
# themed.py -- Example of a TraitsUI with themes
from traits.api import HasTraits, Str, Range, Float, Enum
from traitsui.api import View, Group, Item, Label
from traitsui.wx.themed_text_editor import \
    ThemedTextEditor

class Test ( HasTraits ):

    name   = Str
    age    = Range( 1, 100 )
    weight = Float
    gender = Enum( 'Male', 'Female' )

    view = View(
        Group(
            Group(
                Label( 'A Themed Label', '@GF6' ),
                Item( 'name' ),
                Item( 'age' ),
                Item( 'weight', editor=ThemedTextEditor()),
                Item( 'gender' ),
                group_theme = '@GD0'
            ),
            group_theme = '@G',
            item_theme  = '@B0B',
            label_theme = '@BEA'
        ),
        title   = 'Themed TraitsUI',
    )

Test().configure_traits()
```

This example uses the following theme-related items:

- The **group_theme**, **item_theme**, and **label_theme** attributes are explicitly specified (lines 24 to 26).
- The Label constructor (line 17)takes an optional second argument (in this case '@GF6'), which specifies the **item_theme** information for the Label object. (Label is a subclass of Item.)
- The item for weight (line 20) uses a ThemedTextEditor factory; this isn't strictly necessary, but illustrates the use of a themed editor factory. For more information on themed editor factories, refer to *"Extra" Trait Editor Factories*, and to the *Traits API Reference*.
- The example contains an extra Group level (line 16), and shows the results of two nested **group_theme** values ('@G' and '@GD0'). The outermost **group_theme** value ('@G') specifies the gray background, while the innermost **group_theme** value ('@GD0') specifies the light gray rectangle drawn over it. This combination demonstrates the automatic compositing of themes, since the rounded rectangle is transparent except where the light gray band appears.

- The theme data strings use the '@' prefix to reference images from the default image library.

1.8 Introduction to Trait Editor Factories

The preceding code samples in this User Manual have been surprisingly simple considering the sophistication of the interfaces that they produce. In particular, no code at all has been required to produce appropriate widgets for the Traits to be viewed or edited in a given window. This is one of the strengths of TraitsUI: usable interfaces can be produced simply and with a relatively low level of UI programming expertise.

An even greater strength lies in the fact that this simplicity does not have to be paid for in lack of flexibility. Where a novice TraitsUI programmer can ignore the question of widgets altogether, a more advanced one can select from a variety of predefined interface components for displaying any given Trait. Furthermore, a programmer who is comfortable both with TraitsUI and with UI programming in general can harness the full power and flexibility of the underlying GUI toolkit from within TraitsUI.

The secret behind this combination of simplicity and flexibility is a TraitsUI construct called a trait *editor factory*. A trait editor factory encapsulates a set of display instructions for a given *trait type*, hiding GUI-toolkit-specific code inside an abstraction with a relatively straightforward interface. Furthermore, every *predefined trait type* in the Traits package has a predefined trait editor factory that is automatically used whenever the trait is displayed, unless you specify otherwise.

Consider the following script and the window it creates:

Example 12: Using default trait editors

```
# default_trait_editors.py -- Example of using default
#                             trait editors

from traits.api import HasTraits, Str, Range, Bool
from traitsui.api import View, Item

class Adult(HasTraits):
    first_name = Str
    last_name = Str
    age = Range(21,99)
    registered_voter = Bool


    traits_view = View(Item(name='first_name'),
                       Item(name='last_name'),
                       Item(name='age'),
                       Item(name='registered_voter'))

alice = Adult(first_name='Alice',
              last_name='Smith',
              age=42,
              registered_voter=True)

alice.configure_traits()
```

Notice that each trait is displayed in an appropriate widget, even though the code does not explicitly specify any widgets at all. The two Str traits appear in text boxes, the Range is displayed using a combination of a text box and a slider, and the Bool is represented by a checkbox. Each implementation is generated by the default trait editor factory (TextEditor, RangeEditor and BooleanEditor respectively) associated with the trait type.

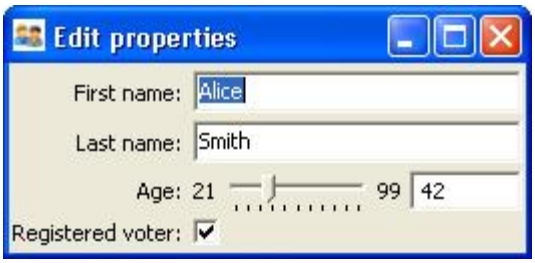


Fig. 1.12: Figure 12: User interface for Example 12

TraitsUI is by no means limited to these defaults. There are two ways to override the default representation of a *trait attribute* in a TraitsUI window:

- Explicitly specifying an alternate trait editor factory
- Specifying an alternate style for the editor generated by the factory

The remainder of this chapter examines these alternatives more closely.

1.8.1 Specifying an Alternate Trait Editor Factory

As of this writing the TraitsUI package includes a wide variety of predefined trait editor factories, which are described in *Basic Trait Editor Factories* and *Advanced Trait Editors*. Some additional editor factories are specific to the wxWidgets toolkit and are defined in one of the following packages:

- traitsui.wx
- traitsui.wx.extra
- traitsui.wx.extra.windows (specific to Microsoft Windows)

These editor factories are described in *"Extra" Trait Editor Factories*.

For a current complete list of editor factories, refer to the *Traits API Reference*.

Other packages can define their own editor factories for their own traits. For example, enthought.kiva.api.KivaFont uses a KivaFontEditor() and enthought.enable2.traits.api.RGBAColor uses an RGBAColorEditor().

For most *predefined trait type*s (see Traits User Manual), there is exactly one predefined trait editor factory suitable for displaying it: the editor factory that is assigned as its default. [15] There are exceptions, however; for example, a Str trait defaults to using a TextEditor, but can also use a CodeEditor or an HTMLEditor. A List trait can be edited by means of ListEditor, TableEditor (if the List elements are HasTraits objects), CheckListEditor or SetEditor. Furthermore, the TraitsUI package includes tools for building additional trait editors and factories for them as needed.

To use an alternate editor factory for a trait in a TraitsUI window, you must specify it in the View for that window. This is done at the Item level, using the *editor* keyword parameter. The syntax of the specification is `editor = editor_factory()`. (Use the same syntax for specifying that the default editor should be used, but with certain keyword parameters explicitly specified; see *Initializing Editors*).

For example, to display a Str trait called **my_string** using the default editor factory (TextEditor()), the View might contain the following Item:

```
Item(name='my_string')
```

The resulting widget would have the following appearance:

To use the HTMLEditor factory instead, add the appropriate specification to the Item:

[15] Appendix II contains a table of the predefined trait types in the Traits package and their default trait editor types.

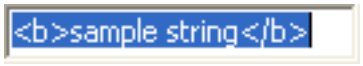


Fig. 1.13: Figure 13: Default editor for a Str trait

```
Item( name='my_string', editor=HTMLEditor() )
```

The resulting widget appears as in Figure 14:

Fig. 1.14: Figure 14: Editor generated by HTMLEditor()

Note: TraitsUI does not check editors for appropriateness.

TraitsUI does not police the *editor* argument to ensure that the specified editor is appropriate for the trait being displayed. Thus there is nothing to prevent you from trying to, say, display a Float trait using ColorEditor(). The results of such a mismatch are unlikely to be helpful, and can even crash the application; it is up to the programmer to choose an editor sensibly. *The Predefined Trait Editor Factories* is a useful reference for selecting an appropriate editor for a given task.

It is possible to specify the trait editor for a trait in other ways:

- You can specify a trait editor when you define a trait, by passing the result of a trait editor factory as the *editor* keyword parameter of the callable that creates the trait. However, this approach commingles the *view* of a trait with its *model*.
- You can specify the **editor** attribute of a TraitHandler object. This approach commingles the *view* of a trait with its *controller*.

Use these approaches very carefully, if at all, as they muddle the *MVC* design pattern.

Initializing Editors

Many of the TraitsUI trait editors can be used "straight from the box" as in the example above. There are some editors, however, that must be initialized in order to be useful. For example, a checklist editor (from CheckListEditor()) and a set editor (from SetEditor()) both enable the user to edit a List attribute by selecting elements from a specified set; the contents of this set must, of course, be known to the editor. This sort of initialization is usually performed by means of one or more keyword arguments to the editor factory, for example:

```
Item(name='my_list',editor=CheckListEditor(values=["opt1","opt2","opt3"]))
```

The descriptions of trait editor factories in *The Predefined Trait Editor Factories* include a list of required and optional initialization keywords for each editor.

1.8.2 Specifying an Editor Style

In TraitsUI, any given trait editor can be generated in one or more of four different styles: *simple*, *custom*, *text* or *readonly*. These styles, which are described in general terms below, represent different "flavors" of data display, so that a given trait editor can look completely different in one style than in another. However, different trait editors displayed in the same style (usually) have noticeable characteristics in common. This is useful because editor style,

unlike individual editors, can be set at the Group or View level, not just at the Item level. This point is discussed further in *Using Editor Styles*.

The 'simple' Style

The *simple* editor style is designed to be as functional as possible while requiring minimal space within the window. In simple style, most of the Traits UI editors take up only a single line of space in the window in which they are embedded.

In some cases, such as the text editor and Boolean editor (see *Basic Trait Editor Factories*), the single line is fully sufficient. In others, such as the (plain) color editor and the enumeration editor, a more detailed interface is required; pop-up panels, drop-down lists, or dialog boxes are often used in such cases. For example, the simple version of the enumeration editor for the wxWidgets toolkit looks like this:

Fig. 1.15: Figure 15: Simple style of enumeration editor

However, when the user clicks on the widget, a drop-down list appears:

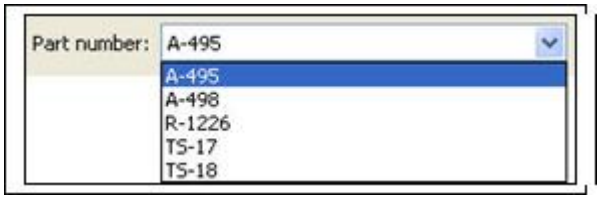


Fig. 1.16: Figure 16: Simple enumeration editor with expanded list

The simple editor style is most suitable for windows that must be kept small and concise.

The 'custom' Style

The *custom* editor style generally generates the most detailed version of any given editor. It is intended to provide maximal functionality and information without regard to the amount of window space used. For example, in the wxWindows toolkit, the custom style the enumeration editor appears as a set of radio buttons rather than a drop-down list:

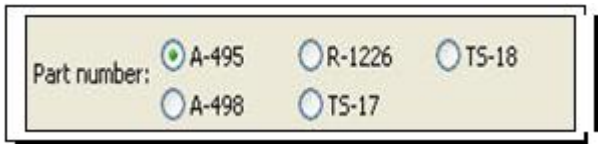


Fig. 1.17: Figure 17: Custom style of enumeration editor

In general, the custom editor style can be very useful when there is no need to conserve window space, as it enables the user to see as much information as possible without having to interact with the widget. It also usually provides the most intuitive interface of the four.

Note that this style is not defined explicitly for all trait editor implementations. If the custom style is requested for an editor for which it is not defined, the simple style is generated instead.

The 'text' Style

The *text* editor style is the simplest of the editor styles. When applied to a given trait attribute, it generates a text representation of the trait value in an editable box. Thus the enumeration editor in text style looks like the following:

Fig. 1.18: Figure 18: Text style of enumeration editor

For this type of editor, the end user must type in a valid value for the attribute. If the user types an invalid value, the validation method for the attribute (see Traits User Manual) notifies the user of the error (for example, by shading the background of the text box red).

The text representation of an attribute to be edited in a text style editor is created in one of the following ways, listed in order of priority:

1. The function specified in the **format_func** attribute of the Item (see *The Item Object*), if any, is called on the attribute value.
2. Otherwise, the function specified in the *format_func* parameter of the trait editor factory, if any, is called on the attribute value.
3. Otherwise, the Python-style formatting string specified in the **format_str** attribute of the Item (see *The Item Object*), if any, is used to format the attribute value.
4. The Python-style formatting string specified in the *format_str* parameter of the trait editor factory, if any, is used to format the attribute value.
5. Otherwise, the Python str() function is called on the attribute value.

The 'readonly' style

The *readonly* editor style is usually identical in appearance to the text style, except that the value appears as static text rather than in an editable box:

Fig. 1.19: Figure 19: Read-only style of enumeration editor

This editor style is used to display data values without allowing the user to change them.

Using Editor Styles

As discussed in *Contents of a View* and *Customizing a View*, the Item, Group and View objects of TraitsUI all have a **style** attribute. The style of editor used to display the Items in a View is determined as follows:

1. The editor style used to display a given Item is the value of its **style** attribute if specifically assigned. Otherwise the editor style of the Group or View that contains the Item is used.
2. The editor style of a Group is the value of its **style** attribute if assigned. Otherwise, it is the editor style of the Group or View that contains the Group.
3. The editor style of a View is the value of its **style** attribute if specified, and 'simple' otherwise.

In other words, editor style can be specified at the Item, Group or View level, and in case of conflicts the style of the smaller scope takes precedence. For example, consider the following script:

Example 13: Using editor styles at various levels

```
# mixed_styles.py -- Example of using editor styles at
#                    various levels

from traits.api import HasTraits, Str, Enum
from traitsui.api import View, Group, Item

class MixedStyles(HasTraits):
   first_name = Str
   last_name = Str

   department = Enum("Business", "Research", "Admin")
   position_type = Enum("Full-Time",
                        "Part-Time",
                        "Contract")

   traits_view = View(Group(Item(name='first_name'),
                            Item(name='last_name'),
                            Group(Item(name='department'),
                                  Item(name='position_type',
                                       style='custom'),
                                  style='simple')),
                      title='Mixed Styles',
                      style='readonly')

ms = MixedStyles(first_name='Sam', last_name='Smith')
ms.configure_traits()
```

Notice how the editor styles are set for each attribute:

- **position_type** at the Item level (lines 19-20)
- **department** at the Group level (lines 18 and 21)
- **first_name** and **last_name** at the View level (lines 16, 17, and 23)

The resulting window demonstrates these precedence rules:

Fig. 1.20: Figure 20: User interface for Example 13

1.9 The Predefined Trait Editor Factories

This chapter contains individual descriptions of the predefined trait editor factories provided by TraitsUI. Most of these editor factories are straightforward and can be used easily with little or no expertise on the part of the programmer or end user; these are described in *Basic Trait Editor Factories*. The section *Advanced Trait Editors* covers a smaller set of specialized editors that have more complex interfaces or that are designed to be used along with complex editors.

Note: Examples are toolkit-specific.

The exact appearance of the editors depends on the underlying GUI toolkit. The screenshots and descriptions in this chapter are based on wxWindows. Another supported GUI toolkit is Qt, from TrollTech.

Rather than trying to memorize all the information in this chapter, you might skim it to get a general idea of the available trait editors and their capabilities, and use it as a reference thereafter.

1.9.1 Basic Trait Editor Factories

The editor factories described in the following sections are straightforward to use. You can pass the editor object returned by the editor factory as the value of the *editor* keyword parameter when defining a trait.

ArrayEditor()

Suitable for 2-D Array, 2-D CArray

Default for Array, CArray (if 2-D)

Optional parameter *width*

The editors generated by ArrayEditor() provide text fields (or static text for the read-only style) for each cell of a two-dimensional Numeric array. Only the simple and read-only styles are supported by the wxWidgets implementation. You can specify the width of the text fields with the *width* parameter.

The following code generates the editors shown in Figure 21.

Example 14: Demonstration of array editors

```
# array_editor.py -- Example of using array editors

import numpy as np
from traits.api import HasPrivateTraits, Array
from traitsui.api \
    import View, ArrayEditor, Item
from traitsui.menu import NoButtons

class ArrayEditorTest ( HasPrivateTraits ):

    three = Array(np.int, (3,3))
    four  = Array(np.float,
                  (4,4),
                  editor = ArrayEditor(width = -50))

    view = View( Item('three', label='3x3 Integer'),
                 '_',
                 Item('three',
```

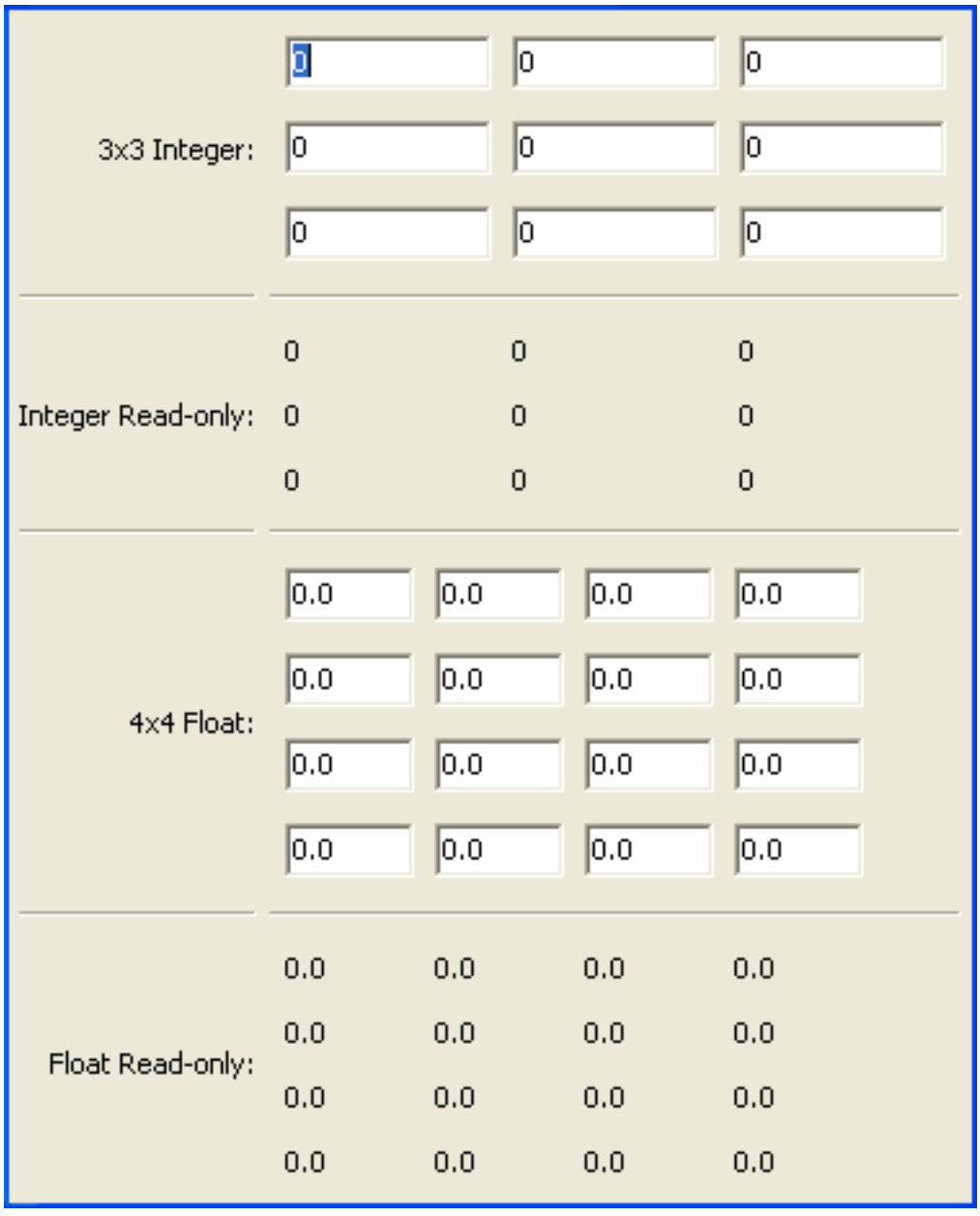


Fig. 1.21: Figure 21: Array editors

```
                        label='Integer Read-only',
                        style='readonly'),
                   '_',
                   Item('four', label='4x4 Float'),
                   '_',
                   Item('four',
                        label='Float Read-only',
                        style='readonly'),
                   buttons   = NoButtons,
                   resizable = True )


if __name__ == '__main__':
    ArrayEditorTest().configure_traits()
```

BooleanEditor()

Suitable for Bool, CBool

Default for Bool, CBool

Optional parameters *mapping*

BooleanEditor is one of the simplest of the built-in editor factories in the TraitsUI package. It is used exclusively to edit and display Boolean (i.e, True/False) traits. In the simple and custom styles, it generates a checkbox. In the text style, the editor displays the trait value (as one would expect) as the strings True or False. However, several variations are accepted as input:

- 'True'
- T
- Yes
- y
- 'False'
- F
- No
- n

The set of acceptable text inputs can be changed by setting the BooleanEditor() parameter *mapping* to a dictionary whose entries are of the form *str*: *val*, where *val* is either True or False and *str* is a string that is acceptable as text input in place of that value. For example, to create a Boolean editor that accepts only yes and no as appropriate text values, you might use the following expression:

```
editor=BooleanEditor(mapping={"yes":True, "no":False})
```

Note that in this case, the strings True and False would *not* be acceptable as text input.

Figure 22 shows the four styles generated by BooleanEditor().

ButtonEditor()

Suitable for Button, Event, ToolbarButton

Default for Button, ToolbarButton

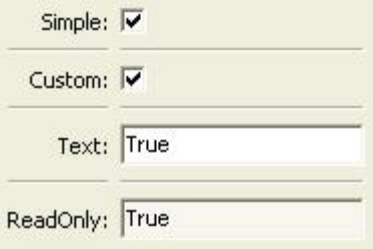


Fig. 1.22: Figure 22: Boolean editor styles

Optional parameters *height_padding*, *image*, *label*, *label_value*, *orientation*, *style*, *value*, *values_trait*, *view*, *width_padding*

The ButtonEditor() factory is designed to be used with an Event or Button [16] trait. When a user clicks a button editor, the associated event is fired. Because events are not printable objects, the text and read-only styles are not implemented for this editor. The simple and custom styles of this editor are identical.

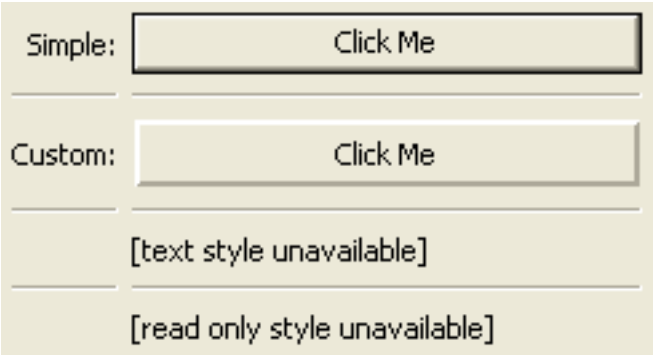


Fig. 1.23: Figure 23: Button editor styles

By default, the label of the button is the name of the Button or Event trait to which it is linked. [17] However, this label can be set to any string by specifying the *label* parameter of ButtonEditor() as that string.

Alternatively, use *label_value* to specify the name of the trait to use as the button label.

You can specify a value for the trait to be set to, using the *value* parameter. If the trait is an Event, then the value is not stored, but might be useful to an event listener.

Use *values_trait* to specify the name of the trait on the object that contains the list of possible values. If this is set, then the *value*, *label*, and *label_value* traits are ignored; instead, they will be set from this list. When this button is clicked, the value set will be the one selected from the drop-down.

CheckListEditor()

Suitable for List

Default for (none)

Optional parameters *cols*, *name*, *values*

[16] In Traits, a Button and an Event are essentially the same thing, except that Buttons are automatically associated with button editors.

[17] TraitsUI makes minor modifications to the name, capitalizing the first letter and replacing underscores with spaces, as in the case of a default Item label (see *The View Object*).

The editors generated by the CheckListEditor() factory are designed to enable the user to edit a List trait by selecting elements from a "master list", i.e., a list of possible values. The list of values can be supplied by the trait being edited, or by the *values* parameter.

The *values* parameter can take either of two forms:

- A list of strings
- A list of tuples of the form (*element*, *label*), where *element* can be of any type and *label* is a string.

In the latter case, the user selects from the labels, but the underlying trait is a List of the corresponding *element* values.

Alternatively, you can use the *name* parameter to specify a trait attribute containing the label strings for the values.

The custom style of editor from this factory is displayed as a set of checkboxes. By default, these checkboxes are displayed in a single column; however, you can initialize the *cols* parameter of the editor factory to any value between 1 and 20, in which case the corresponding number of columns is used.

The simple style generated by CheckListEditor() appears as a drop-down list; in this style, only one list element can be selected, so it returns a list with a single item. The text and read-only styles represent the current contents of the attribute in Python-style text format; in these cases the user cannot see the master list values that have not been selected.

The four styles generated by CheckListEditor() are shown in Figure 24. Note that in this case the *cols* parameter has been set to 4.

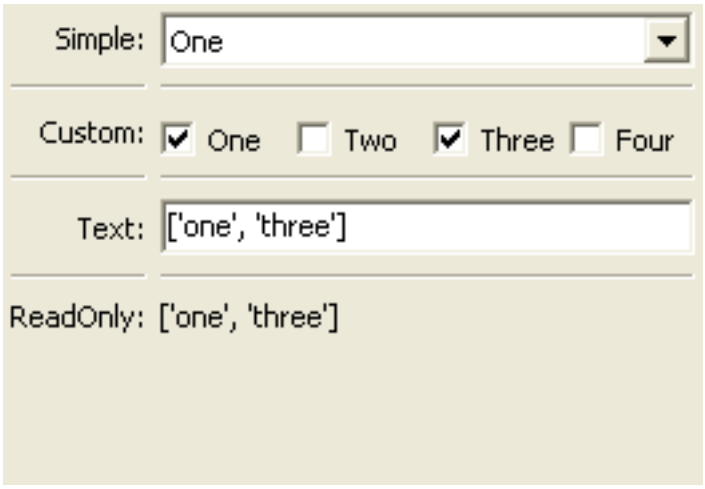


Fig. 1.24: Figure 24: Checklist editor styles

CodeEditor()

Suitable for Code, Str, String

Default for Code

Optional parameters *auto_set*

The purpose of a code editor is to display and edit Code traits, though it can be used with the Str and String trait types as well. In the simple and custom styles (which are identical for this editor), the text is displayed in numbered, non-wrapping lines with a horizontal scrollbar. The text style displays the trait value using a single scrolling line with special characters to represent line breaks. The read-only style is similar to the simple and custom styles except that the text is not editable.

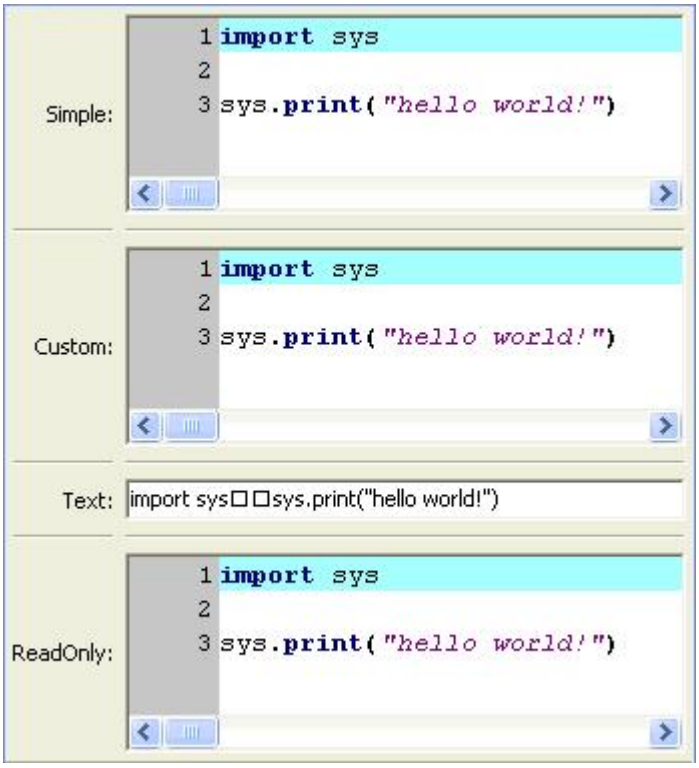


Fig. 1.25: Figure 25: Code editor styles

The *auto_set* keyword parameter is a Boolean value indicating whether the trait being edited should be updated with every keystroke (True) or only when the editor loses focus, i.e., when the user tabs away from it or closes the window (False). The default value of this parameter is True.

ColorEditor()

Suitable for Color

Default for Color

Optional parameters *mapped*

The editors generated by ColorEditor() are designed to enable the user to display a Color trait or edit it by selecting a color from the palette available in the underlying GUI toolkit. The four styles of color editor are shown in Figure 26.

In the simple style, the editor appears as a text box whose background is a sample of the currently selected color. The text in the box is either a color name or a tuple of the form (*r*, *g*, *b*) where *r*, *g*, and *b* are the numeric values of the red, green and blue color components respectively. (Which representation is used depends on how the value was entered.) The text value is not directly editable in this style of editor; instead, clicking on the text box displays a pop-up panel similar in appearance and function to the custom style.

The custom style includes a labeled color swatch on the left, representing the current value of the Color trait, and a palette of common color choices on the right. Clicking on any tile of the palette changes the color selection, causing the swatch to update accordingly. Clicking on the swatch itself causes a more detailed, platform-specific interface to appear in a dialog box, such as is shown in Figure 27.

The text style of editor looks exactly like the simple style, but the text box is editable (and clicking on it does not open a pop-up panel). The user must enter a recognized color name or a properly formatted (*r*, *g*, *b*) tuple.

Fig. 1.26: Figure 26: Color editor styles

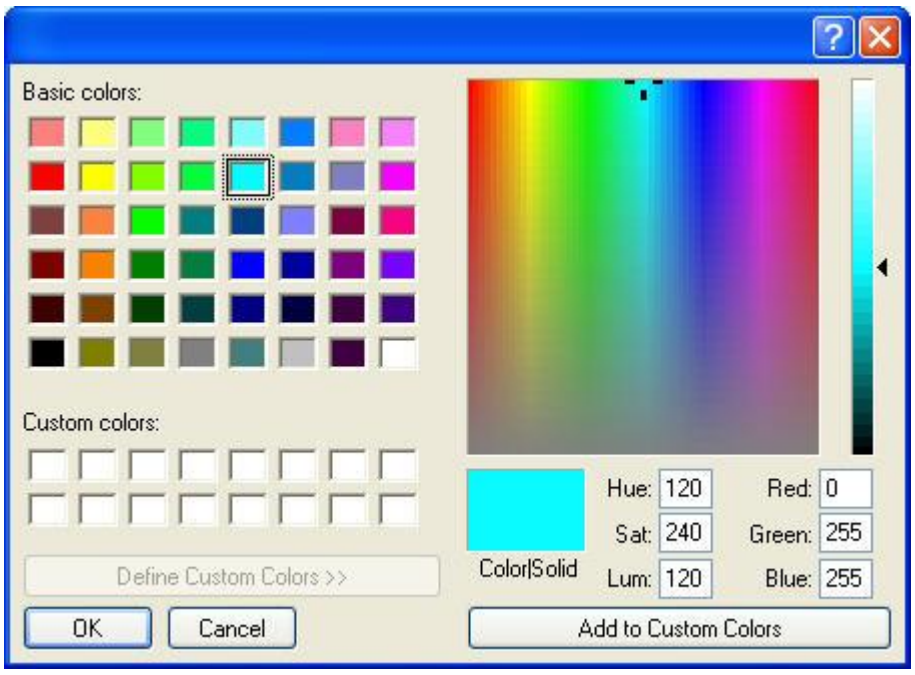


Fig. 1.27: Figure 27: Custom color selection dialog box for Microsoft Windows XP

The read-only style displays the text representation of the currently selected Color value (name or tuple) on a minimally-sized background of the corresponding color.

For advanced users: The *mapped* keyword parameter of ColorEditor() is a Boolean value indicating whether the trait being edited has a built-in mapping of user-oriented representations (e.g., strings) to internal representations. Since ColorEditor() is generally used only for Color traits, which are mapped (e.g., 'cyan' to wx.Colour(0,255,255)), this parameter defaults to True and is not of interest to most programmers. However, it is possible to define a custom color trait that uses ColorEditor() but is not mapped (i.e., uses only one representation), which is why the attribute is available.

CompoundEditor()

Suitable for special

Default for "compound" traits

Optional parameters *auto_set*

An editor generated by CompoundEditor() consists of a combination of the editors for trait types that compose the compound trait. The widgets for the compound editor are of the style specified for the compound editor (simple, custom, etc.). The editors shown in Figure 28 are for the following trait, whose value can be an integer between 1 and 6, or any of the letters 'a' through 'f':

```
compound_trait = Trait( 1, Range( 1, 6 ), 'a', 'b', 'c', 'd', 'e', 'f')
```

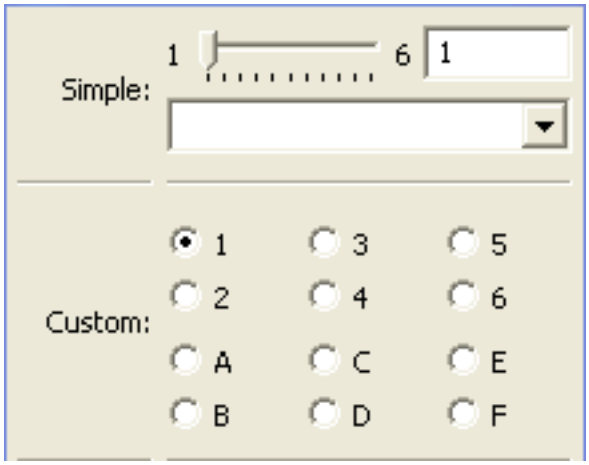


Fig. 1.28: Figure 28: Example compound editor styles

The *auto_set* keyword parameter is a Boolean value indicating whether the trait being edited should be updated with every keystroke (True) or only when the editor loses focus, i.e., when the user tabs away from it or closes the window (False). The default value of this parameter is True.

CSVListEditor()

Suitable for lists of simple data types

Default for none

Optional parameters *auto_set*, *enter_set*, *ignore_trailing_sep*, *sep*

This editor provides a line of text for editing a list of certain simple data types. The following List traits can be edited by a CSVListEditor:

- List(Int)
- List(Float)
- List(Str)
- List(Enum('string1', 'string2', *etc*))
- List(Range(low= *low value or trait name*, high= *high value or trait name*))

The 'text', 'simple' and 'custom' styles are all the same. They provide a single line of text in which the user can enter the list. The 'readonly' style provides a line of text that can not be edited by the user.

The default separator of items in the list is a comma. This can be overridden with the *sep* keyword parameter.

Parameters

auto_set [bool] If *auto_set* is True, each key pressed by the user triggers validation of the input, and if it is valid, the value of the object being edited is updated. *Default:* True

enter_set [bool] If *enter_set* is True, the input is updated when the user presses the *Enter* key. *Default:* False

sep [str or None] The separator of the list item in the text field. If *sep* is None, each contiguous span of whitespace is a separator. (Note: After the text field is split at the occurrences of *sep*, leading and trailing whitespace is removed from each item before converting to the underlying data type.) *Default:* ',' (a comma)

ignore_trailing_sep [bool] If *ignore_trailing_sep* is True, the user may enter a trailing separator (e.g. '1, 2, 3,') and it will be ignored. If this is False, a trailing separator is an error. *Default:* True

See Also

ListEditor, TextEditor

DefaultOverride()

Suitable for (any)

Default for (none)

The DefaultOverride() is a factory that takes the trait's default editor and customizes it with the specified parameters. This is useful when a trait defines a default editor using some of its data, e.g. Range or Enum, and you want to tweak some of the other parameters without having recreate that data.

For example, the default editor for Range(low=0, high=1500) has '1500' as the upper label. To change it to 'Max' instead, use:

```
View(Item('my_range', editor=DefaultOverride(high_label='Max'))
```

DirectoryEditor()

Suitable for Directory

Default for Directory

Optional parameters *entries*, *filter*, *filter_name*, *reload_name*, *truncate_ext*, *dclick_name*

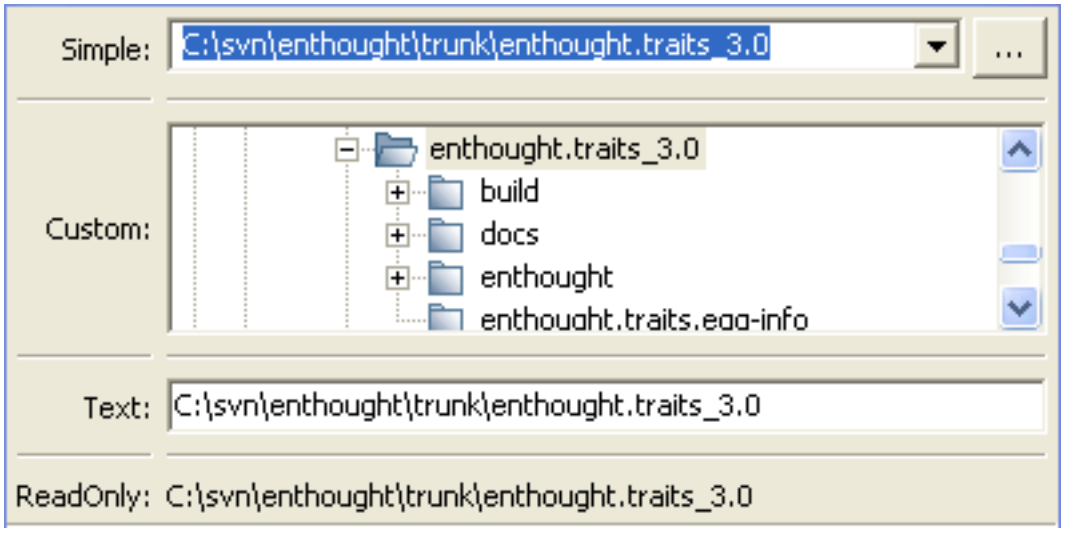


Fig. 1.29: Figure 29: Directory editor styles

A directory editor enables the user to display a Directory trait or set it to some directory in the local system hierarchy. The four styles of this editor are shown in Figure 29.

In the simple style, the current value of the trait is displayed in a combo box to the left of a button labeled '...'. The user can type a new path directly into the text box, select a previous value from the droplist of the combo box, or use the button to bring up a directory browser panel similar to the custom style of editor.

When the user selects a directory in this browser, the panel collapses, and control is returned to the original editor widget, which is automatically populated with the new path string.

The user can also drag and drop a directory object onto the simple style editor.

The custom style displays a directory browser panel, in which the user can expand or collapse directory structures, and click a folder icon to select a directory.

The text style of editor is simply a text box into which the user can type a directory path. The 'readonly' style is identical to the text style, except that the text box is not editable.

The optional parameters are the same as the FileEditor.

No validation is performed on Directory traits; the user must ensure that a typed-in value is in fact an actual directory on the system.

EnumEditor()

Suitable for Enum, Any

Default for Enum

Required parameters for non-Enum traits: *values* or *name*

Optional parameters *cols*, *evaluate*, *mode*

The editors generated by EnumEditor() enable the user to pick a single value from a closed set of values.

The simple style of editor is a drop-down list box.

The custom style is a set of radio buttons. Use the *cols* parameter to specify the number of columns of radio buttons.

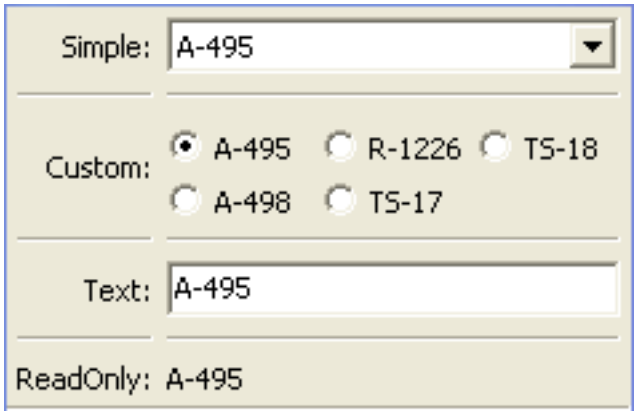


Fig. 1.30: Figure 30: Enumeration editor styles

The text style is an editable text field; if the user enters a value that is not in enumerated set, the background of the field turns red, to indicate an error. You can specify a function to evaluate text input, using the *evaluate* parameter.

The read-only style is the value of the trait as static text.

If the trait attribute that is being edited is not an enumeration, you must specify either the trait attribute (with the *name* parameter), or the set of values to display (with the *values* parameter). The *name* parameter can be an extended trait name. The *values* parameter can be a list, tuple, or dictionary, or a "mapped" trait.

By default, an enumeration editor sorts its values alphabetically. To specify a different order for the items, give it a mapping from the normal values to ones with a numeric tag. The enumeration editor sorts the values based on the numeric tags, and then strips out the tags.

Example 15: Enumeration editor with mapped values

```
# enum_editor.py -- Example of using an enumeration editor
from traits.api import HasTraits, Enum
from traitsui.api import EnumEditor

Class EnumExample(HasTraits):
    priority = Enum('Medium', 'Highest',
                              'High',
                              'Medium',
                              'Low',
                              'Lowest')

    view = View( Item(name='priority',
                      editor=EnumEditor(values={
                          'Highest' : '1:Highest',
                          'High'    : '2:High',
                          'Medium'  : '3:Medium',
                          'Low'     : '4:Low',
                          'Lowest'  : '5:Lowest', })))
```

The enumeration editor strips the characters up to and including the colon. It assumes that all the items have the colon in the same position; therefore, if some of your tags have multiple digits, you should use zeros to pad the items that have fewer digits.

FileEditor()

Suitable for File

Default for File

Optional parameters *entries*, *filter*, *filter_name*, *reload_name*, *truncate_ext*, *dclick_name*

A file editor enables the user to display a File trait or set it to some file in the local system hierarchy. The styles of this editor are shown in Figure 31.

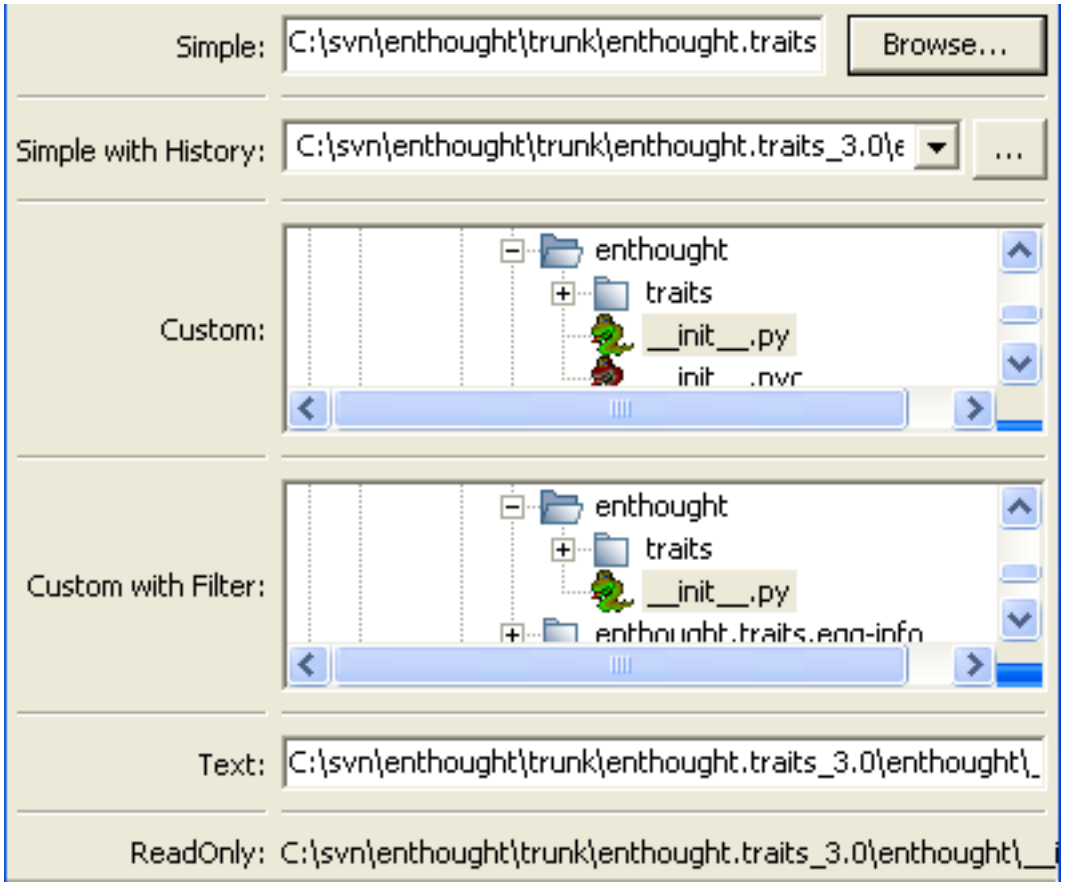


Fig. 1.31: Figure 31: File editor styles

The default version of the simply style displays a text box and a *Browse* button. Clicking *Browse* opens a platform-specific file selection dialog box. If you specify the *entries* keyword parameter with an integer value to the factory function, the simple style is a combo box and a button labeled .... The user can type a file path in the combo box, or select one of *entries* previous values. Clicking the ... button opens a browser panel similar to the custom style of editor. When the user selects a file in this browser, the panel collapses, and control is returned to the original editor widget, which is automatically populated with the new path string.

For either version of the simple style, the user can drag and drop a file object onto the control.

The custom style displays a file system browser panel, in which the user can expand or collapse directory structures, and click an icon to select a file.

You can specify a list of filters to apply to the file names displayed, using the *filter* keyword parameter of the factory function. In Figure 31, the "Custom with Filter" editor uses a *filter* value of `['*.py']` to display only Python source files. You can also specify this parameter for the simple style, and it will be used in the file selection dialog box or

pop-up file system browser panel. Alternatively, you can specify *filter_name*, whose value is an extended trait name of a trait attribute that contains the list of filters.

The *reload_name* parameter is an extended trait name of a trait attribute that is used to notify the editor when the view of the file system needs to be reloaded.

The *truncate_ext* parameter is a Boolean that indicates whether the file extension is removed from the returned filename. It is False by default, meaning that the filename is not modified before it is returned.

The *dclick_name* parameter is an extended trait name of a trait event which is fired when the user double-clicks on a file name when using the custom style.

FontEditor()

Suitable for Font

Default for Font

A font editor enables the user to display a Font trait or edit it by selecting one of the fonts provided by the underlying GUI toolkit. The four styles of this editor are shown in Figure 32.

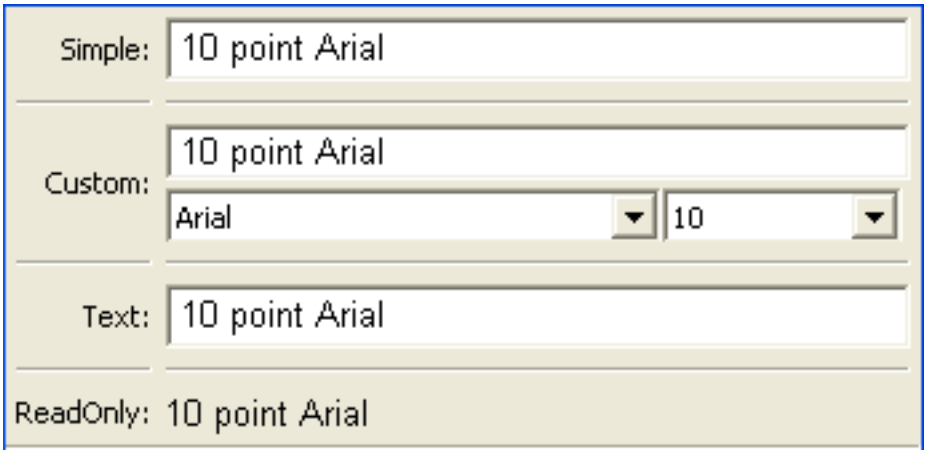


Fig. 1.32: Figure 32: Font editor styles

In the simple style, the currently selected font appears in a display similar to a text box, except that when the user clicks on it, a platform-specific dialog box appears with a detailed interface, such as is shown in Figure 33. When the user clicks *OK*, control returns to the editor, which then displays the newly selected font.

In the custom style, an abbreviated version of the font dialog box is displayed in-line. The user can either type the name of the font in the text box or use the two drop-down lists to select a typeface and size.

In the text style, the user *must* type the name of a font in the text box provided. No validation is performed; the user must enter the correct name of an available font. The read-only style is identical except that the text is not editable.

HTMLEditor()

Suitable for HTML, string traits

Default for HTML

Optional parameters *format_text*

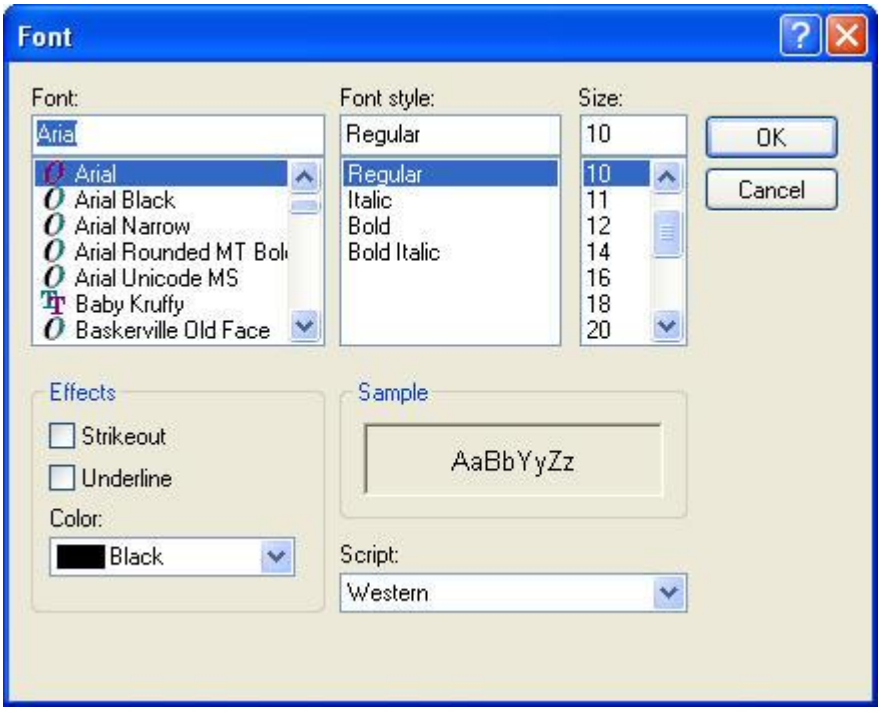


Fig. 1.33: Figure 33: Example font dialog box for Microsoft Windows

The "editor" generated by HTMLEditor() interprets and displays text as HTML. It does not support the user editing the text that it displays. It generates the same type of editor, regardless of the style specified. Figure 34 shows an HTML editor in the upper pane, with a code editor in the lower pane, displaying the uninterpreted text.

Note: HTML support is limited in the wxWidgets toolkit.

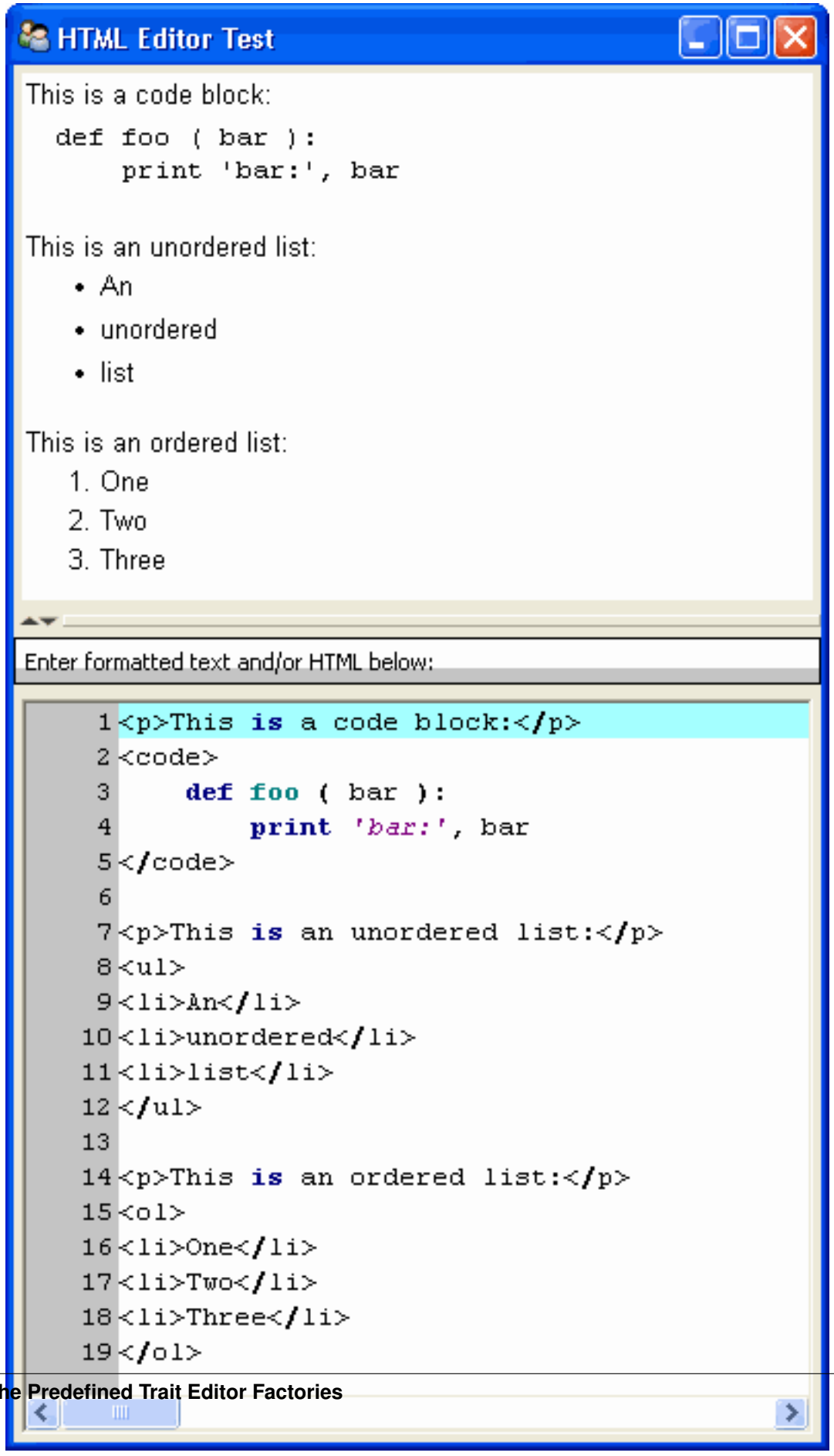
The set of tags supported by the wxWidgets implementation of the HTML editor is a subset of the HTML 3.2 standard. It does not support style sheets or complex formatting. Refer to the wxWidgets documentation for details.

If the *format_text* argument is True, then the HTML editor supports basic implicit formatting, which it converts to HTML before passing the text to the HTML interpreter. The implicit formatting follows these rules:

- Indented lines that start with a dash ('-') are converted to unordered lists.
- Indented lines that start with an asterisk ('*') are converted to ordered lists.
- Indented lines that start with any other character are converted to code blocks.
- Blank lines are converted to paragraph separators.

The following text produces the same displayed HTML as in Figure 34, when *format_text* is True:

```
This is a code block:

    def foo ( bar ):
        print 'bar:', bar

This is an unordered list:
 - An
 - unordered
 - list
```


HTML Editor Test
This is a code block:
def foo (bar):
print 'bar:', bar
This is an unordered list:
An
unordered
list
This is an ordered list:
One
Two
Three
Enter formatted text and/or HTML below:
This is a code block:
<code>
def foo (bar):
print 'bar:', bar
</code>
This is an unordered list:

An
unordered
list

This is an ordered list:

One
Two
Three


```
This is an ordered list:
 * One
 * Two
 * Three
```

ImageEnumEditor()

Suitable for Enum, Any

Default for (none)

Required parameters for non-Enum traits: *values* or *name*

Optional parameters *path*, *klass* or *module*, *cols*, *evaluate*, *suffix*

The editors generated by ImageEnumEditor() enable the user to select an item in an enumeration by selecting an image that represents the item.

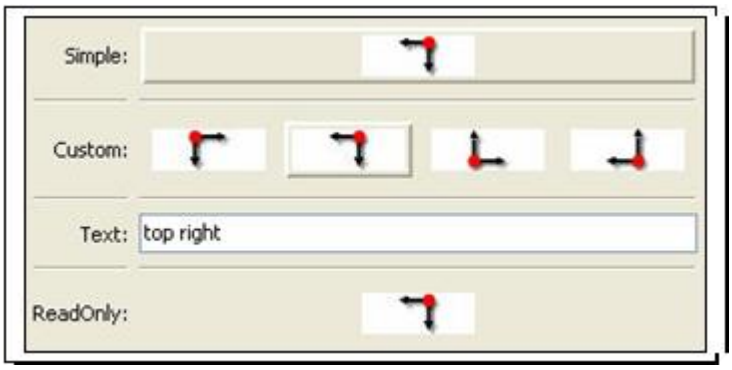


Fig. 1.35: Figure 35: Editor styles for image enumeration

The custom style of editor displays a set of images; the user selects one by clicking it, and it becomes highlighted to indicate that it is selected.

The simple style displays a button with an image for the currently selected item. When the user clicks the button, a pop-up panel displays a set of images, similar to the custom style. The user clicks an image, which becomes the new image on the button.

The text style does not display images; it displays the text representation of the currently selected item. The user must type the text representation of another item to select it.

The read-only style displays the image for the currently selected item, which the user cannot change.

The ImageEnumEditor() function accepts the same parameters as the EnumEditor() function (see *EnumEditor()*), as well as some additional parameters.

Note: Image enumeration editors do not use ImageResource.

Unlike most other images in the Traits and TraitsUI packages, images in the wxWindows implementation of image enumeration editors do not use the PyFace ImageResource class.

In the wxWidgets implementation, image enumeration editors use the following rules to locate images to use:

1. Only GIF (.gif) images are currently supported.

2. The base file name of the image is the string representation of the value, with spaces replaced by underscores and the suffix argument, if any, appended. Note that suffix is not a file extension, but rather a string appended to the base file name. For example, if *suffix* is `_origin` and the *value* is 'top left', the image file name is `top_left_origin.gif`.
3. If the *path* parameter is defined, it is used to locate the file. It can be absolute or relative to the file where the image enumeration editor is defined.
4. If *path* is not defined and the *klass* parameter is defined, it is used to locate the file. The *klass* parameter must be a reference to a class. The editor searches for an images subdirectory in the following locations:
   (a) The directory that contains the module that defines the class.
   (b) If the class was executed directly, the current working directory.
   (c) If *path* and *klass* are not defined, and the *module* parameter is defined, it is used to locate the file. The *module* parameter must be a reference to a module. The editor searches for an images subdirectory of the directory that contains the module.
   (d) If *path*, *klass*, and *module* are not defined, the editor searches for an images subdirectory of the traitsui.wx package.
   (e) If none of the above paths are defined, the editor searches for an `images` directory that is a sibling of the directory from which the application was run.

InstanceEditor()

Suitable for Instance, Property, self, ThisClass, This

Default for Instance, self, ThisClass, This

Optional parameters *cachable*, *editable*, *id*, *kind*, *label*, *name*, *object*, *orientation*, *values*, *view*

The editors generated by InstanceEditor() enable the user to select an instance, or edit an instance, or both.

Editing a Single Instance

In the simplest case, the user can modify the trait attributes of an instance assigned to a trait attribute, but cannot modify which instance is assigned.

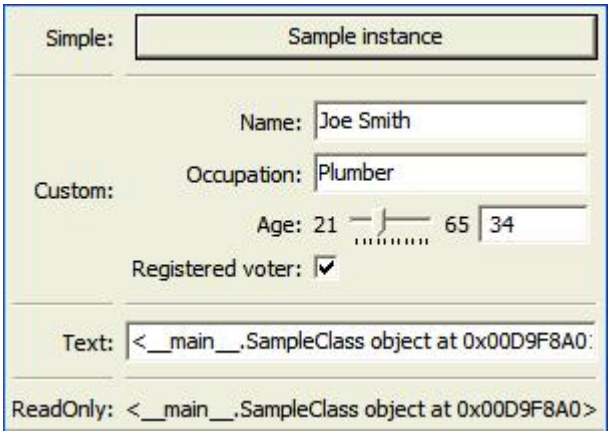


Fig. 1.36: Figure 36: Editor styles for instances

The custom style displays a user interface panel for editing the trait attributes of the instance. The simple style displays a button, which when clicked, opens a window containing a user interface for the instance. The *kind* parameter specifies the kind of window to open (see *Stand-alone Windows*). The *label* parameter specifies a label for the button in the simple interface. The *view* parameter specifies a view to use for the referenced instance's user interface; if this is not specified, the default view for the instance is used (see *Defining a Default View*).

The text and read-only styles display the string representation of the instance. They therefore cannot be used to modify the attributes of the instance. A user could modify the assigned instance if they happened to know the memory address of another instance of the same type, which is unlikely. These styles can useful for prototyping and debugging, but not for real applications.

Selecting Instances

You can add an option to select a different instance to edit. Use the *name* parameter to specify the extended name of a trait attribute in the context that contains a list of instances that can be selected or edited. (See *The View Context* for an explanation of contexts.) Using these parameters results in a drop-drown list box containing a list of text representations of the available instances. If the instances have a **name** trait attribute, it is used for the string in the list; otherwise, a user-friendly version of the class name is used.

For example, the following code defines a Team class and a Person class. A Team has a roster of Persons, and a captain. In the view for a team, the user can pick a captain and edit that person's information. Example 16: Instance editor with instance selection

```
# instance_editor_selection.py -- Example of an instance editor
#                                 with instance selection

from traits.api    \
    import HasStrictTraits, Int, Instance, List, Regex, Str
from traitsui.api \
    import View, Item, InstanceEditor

class Person ( HasStrictTraits ):
    name  = Str
    age   = Int
    phone = Regex( value = '000-0000',
                   regex = '\d\d\d[-]\d\d\d\d' )

    traits_view = View( 'name', 'age', 'phone' )

people = [
  Person( name = 'Dave',   age = 39, phone = '555-1212' ),
  Person( name = 'Mike',   age = 28, phone = '555-3526' ),
  Person( name = 'Joe',    age = 34, phone = '555-6943' ),
  Person( name = 'Tom',    age = 22, phone = '555-7586' ),
  Person( name = 'Dick',   age = 63, phone = '555-3895' ),
  Person( name = 'Harry',  age = 46, phone = '555-3285' ),
  Person( name = 'Sally',  age = 43, phone = '555-8797' ),
  Person( name = 'Fields', age = 31, phone = '555-3547' )
]

class Team ( HasStrictTraits ):

    name    = Str
    captain = Instance( Person )
    roster  = List( Person )

    traits_view = View( Item('name'),
```

```
                        Item('_'),
                        Item( 'captain',
                              label='Team Captain',
                              editor =
                                  InstanceEditor( name = 'roster',
                                                  editable = True),
                              style = 'custom',
                             ),
                        buttons = ['OK'])

if __name__ == '__main__':
    Team( name    = 'Vultures',
          captain = people[0],
          roster  = people ).configure_traits()
```

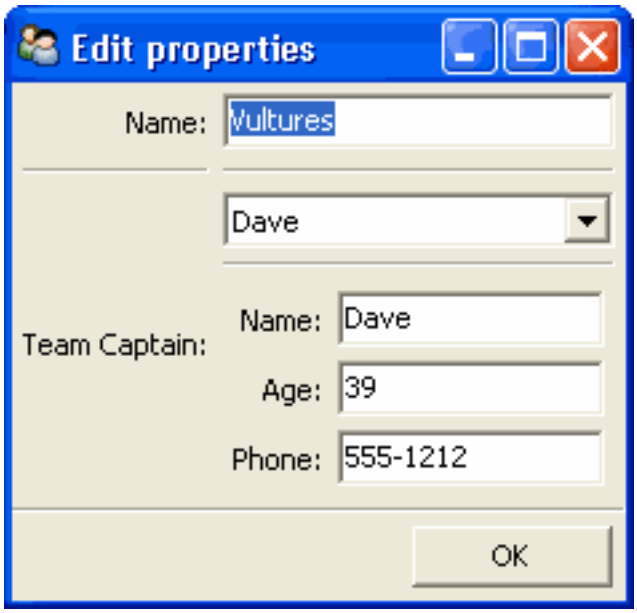


Fig. 1.37: Figure 37: User interface for Example 16

If you want the user to be able to select instances, but not modify their contents, set the *editable* parameter to False. In that case, only the selection list for the instances appears, without the user interface for modifying instances.

Allowing Instances

You can specify what types of instances can be edited in an instance editor, using the *values* parameter. This parameter is a list of items describing the type of selectable or editable instances. These items must be instances of subclasses of traitsui.api.InstanceChoiceItem. If you want to generate new instances, put an InstanceFactoryChoice instance in the *values* list that describes the instance to create. If you want certain types of instances to be dropped on the editor, use an InstanceDropChoice instance in the values list.

ListEditor()

Suitable for List

Default for List [18]

Optional parameters *editor*, *rows*, *style*, *trait_handler*, *use_notebook*

The following parameters are used only if *use_notebook* is True: *deletable*, *dock_style*, *export*, *page_name*, *select*, *view*

The editors generated by ListEditor() enable the user to modify the contents of a list, both by editing the individual items and by adding, deleting, and reordering items within the list.

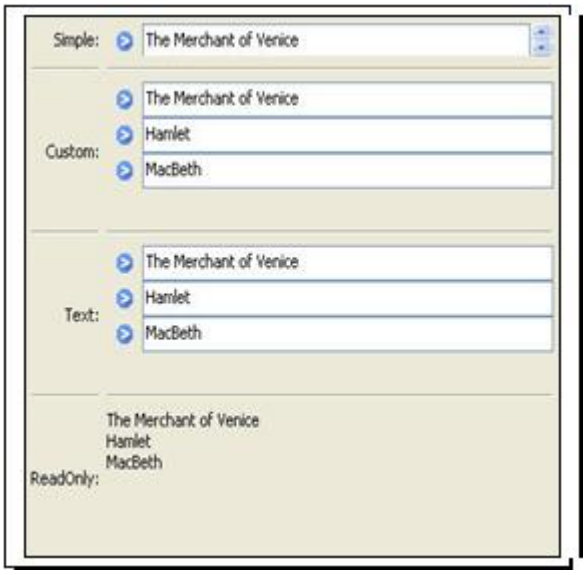


Fig. 1.38: Figure 38: List editor styles

The simple style displays a single item at a time, with small arrows on the right side to scroll the display. The custom style shows multiple items. The number of items displayed is controlled by the *rows* parameter; if the number of items in the list exceeds this value, then the list display scrolls. The editor used for each item in the list is determined by the *editor* and *style* parameters. The text style of list editor is identical to the custom style, except that the editors for the items are text editors. The read-only style displays the contents of the list as static text.

By default, the items use the trait handler appropriate to the type of items in the list. You can specify a different handler to use for the items using the *trait_handler* parameter.

For the simple, custom, and text list editors, a button appears to the left of each item editor; clicking this button opens a context menu for modifying the list, as shown in Figure 39.

In addition to the four standard styles for list editors, a fifth list editor user interface option is available. If *use_notebook* is True, then the list editor displays the list as a "notebook" of tabbed pages, one for each item in the list, as shown in Figure 40. This style can be useful in cases where the list items are instances with their own views. If the *deletable* parameter is True, a close box appears on each tab, allowing the user to delete the item; the user cannot add items interactively through this style of editor.

ListStrEditor()

Suitable for ListStr or List of values mapped to strings

Default for (none)

[18] If a List is made up of HasTraits objects, a table editor is used instead; see *TableEditor()*.

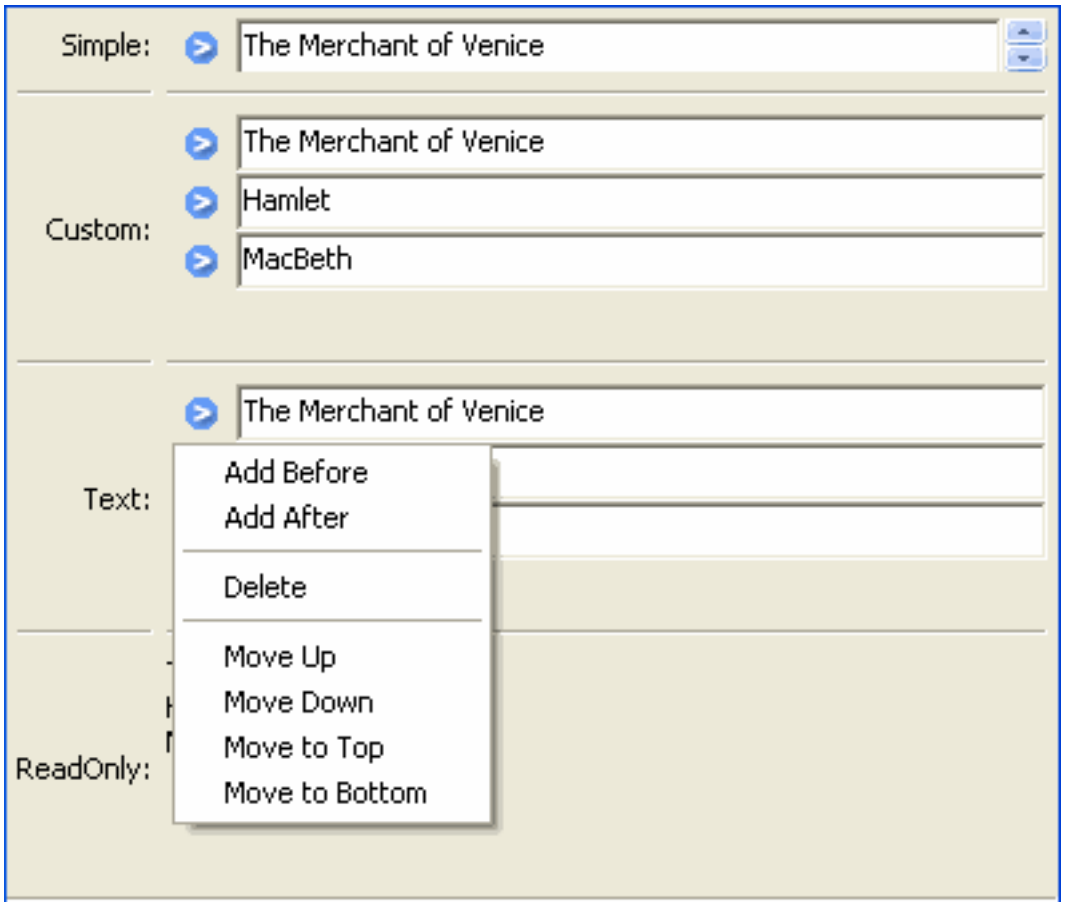


Fig. 1.39: Figure 39: List editor showing context menu

Fig. 1.40: Figure 40: Notebook list editor

Optional parameters *activated, activated_index, adapter, adapter_name, auto_add, drag_move*, *editable, horizontal_lines, images, multi_select, operations*, *right_clicked, right_clicked_index, selected, selected_index, title*, *title_name*

ListStrEditor() generates a list of selectable items corresponding to items in the underlying trait attribute. All styles of the editor are the same. The parameters to ListStrEditor() control aspects of the behavior of the editor, such as what operations it allows on list items, whether items are editable, and whether more than one can be selected at a time. You can also specify extended references for trait attributes to synchronize with user actions, such as the item that is currently selected, activated for editing, or right-clicked.

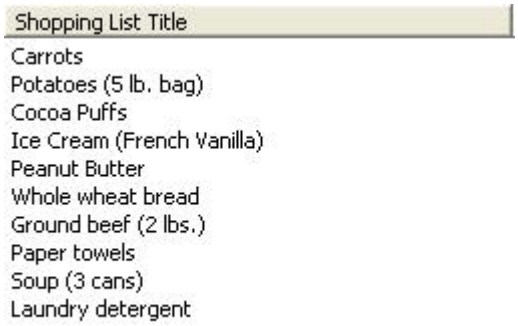


Fig. 1.41: Figure 41: List string editor

NullEditor()

Suitable for controlling layout

Default for (none)

The NullEditor() factory generates a completely empty panel. It is used by the Spring subclass of Item, to generate a blank space that uses all available extra space along its layout orientation. You can also use it to create a blank area of a fixed height and width.

RangeEditor()

Suitable for Range

Default for Range

Optional parameters *auto_set*, *cols*, *enter_set*, *format*, *high_label*, *high_name*, *label_width*, *low_label*, *low_name*, *mode*

The editors generated by RangeEditor() enable the user to specify numeric values within a range. The widgets used to display the range vary depending on both the numeric type and the size of the range, as described in Table 8 and shown in Figure 42. If one limit of the range is unspecified, then a text editor is used.

Table 8: Range editor widgets

Data type/range size	Simple	Custom	Text	Read-only
Integer: Small Range (Size 0-16)	Slider with text box	Radio buttons	Text field	Static text
Integer: Medium Range (Size 17-101)	Slider with text box	Slider with text box	Text field	Static text
Integer: Large Range (Size > 101)	Spin box	Spin box	Text field	Static text
Floating Point: Small Range (Size <= 100.0)	Slider with text box	Slider with text box	Text field	Static text
Floating Point: Large Range (Size > 100.0)	Large-range slider	Large-range slider	Text field	Static text

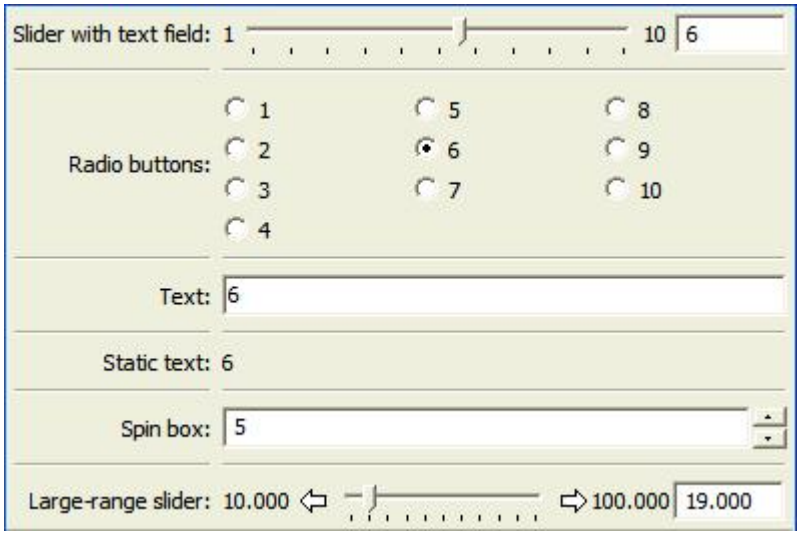


Fig. 1.42: Figure 42: Range editor widgets

In the large-range slider, the arrows on either side of the slider move the editable range, so that the user can move the slider more precisely to the desired value.

You can override the default widget for each type of editor using the *mode* parameter, which can have the following values:

- 'auto': The default widget, as described in Table 8
- 'slider': Simple slider with text field
- 'xslider': Large-range slider with text field
- 'spinner': Spin box with increment/decrement buttons
- 'enum': Radio buttons
- 'text': Text field

You can set the limits of the range dynamically, using the *low_name* and *high_name* parameters to specify trait attributes that contain the low and high limit values; use *low_label*, *high_label* and *label_width* to specify labels for the limits.

RGBColorEditor()

Suitable for RGBColor

Default for RGBColor

Optional parameters *mapped*

Editors generated by RGBColorEditor() are identical in appearance to those generated by ColorEditor(), but they are used for RGBColor traits. See *ColorEditor()* for details.

SetEditor()

Suitable for List

Default for (none)

Required parameters Either *values* or *name*

Optional parameters *can_move_all*, *left_column_title*, *object*, *ordered*, *right_column_title*

In the editors generated by SetEditor(), the user can select a subset of items from a larger set. The two lists are displayed in list boxes, with the candidate set on the left and the selected set on the right. The user moves an item from one set to the other by selecting the item and clicking a direction button (> for left-to-right and < for right-to-left).

Additional buttons can be displayed, depending on two Boolean parameters:

- If *can_move_all* is True, additional buttons appear, whose function is to move all items from one side to the other (>> for left-to-right and << for right-to-left).
- If *ordered* is True, additional buttons appear, labeled *Move up* and *Move down*, which affect the position of the selected item within the set in the right list box.

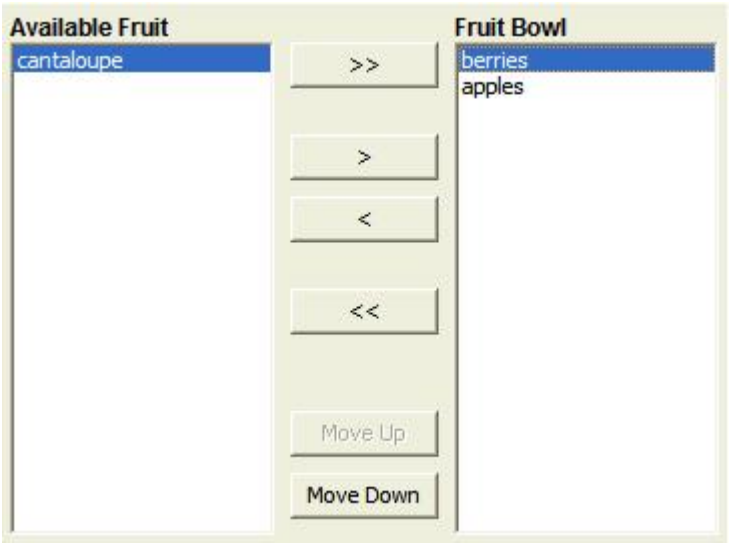


Fig. 1.43: Figure 43: Set editor showing all possible buttons

You can specify the set of candidate items in either of two ways:

- Set the *values* parameter to a list, tuple, dictionary, or mapped trait.
- Set the *name* parameter to the extended name of a trait attribute that contains the list.

ShellEditor()

Suitable for special

Default for PythonValue

The editor generated by ShellEditor() displays an interactive Python shell.

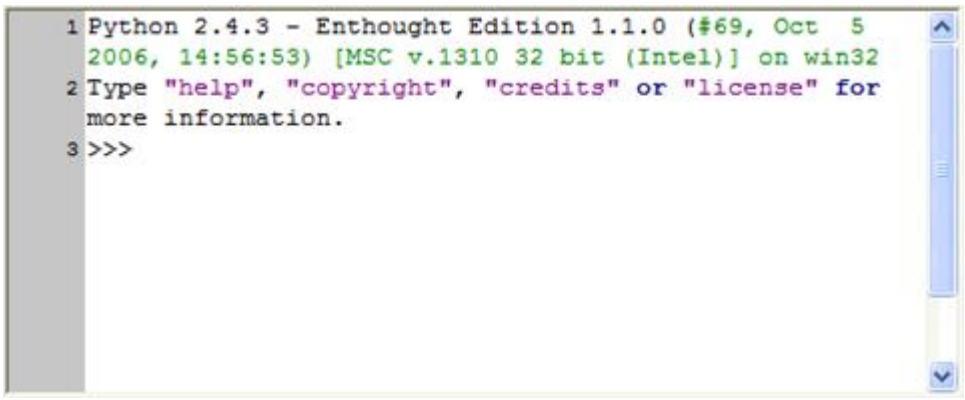

1 Python 2.4.3 - Enthought Edition 1.1.0 (#69, Oct 5
2006, 14:56:53) [MSC v.1310 32 bit (Intel)] on win32
2 Type "help", "copyright", "credits" or "license" for
more information.
3 >>>

Fig. 1.44: Figure 44: Python shell editor

TextEditor()

Suitable for all

Default for Str, String, Password, Unicode, Int, Float, Dict, CStr, CUnicode, and any trait that does not have a specialized TraitHandler

Optional parameters *auto_set*, *enter_set*, *evaluate*, *evaluate_name*, *mapping*, *multi_line*, *password*

The editor generated by TextEditor() displays a text box. For the custom style, it is a multi-line field; for the read-only style, it is static text. If *password* is True, the text that the user types in the text box is obscured.

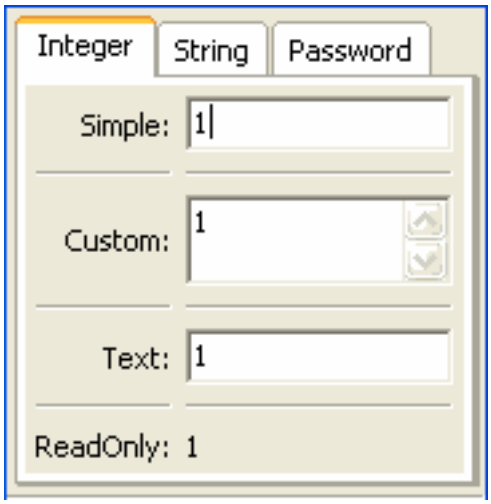


Fig. 1.45: Figure 45: Text editor styles for integers

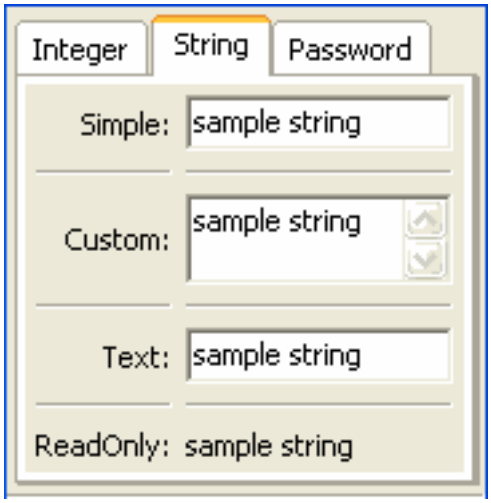


Fig. 1.46: Figure 46: Text editor styles for strings

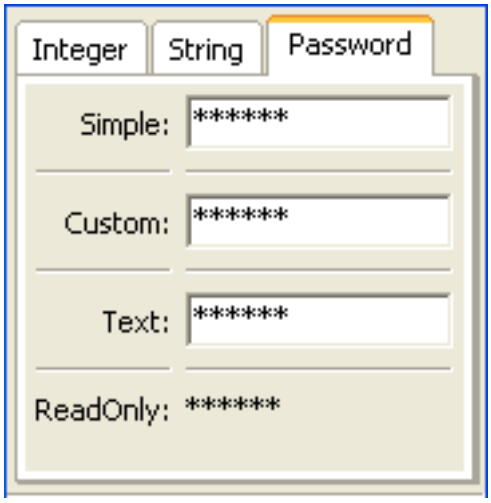


Fig. 1.47: Figure 47: Text editor styles for passwords

You can specify whether the trait being edited is updated on every keystroke (`auto_set=True`) or when the user presses the Enter key (`enter_set=True`). If *auto_set* and *enter_set* are False, the trait is updated when the user shifts the input focus to another widget.

You can specify a mapping from user input values to other values with the *mapping* parameter. You can specify a function to evaluate user input, either by passing a reference to it in the *evaluate* parameter, or by passing the extended name of a trait that references it in the *evaluate_name* parameter.

TitleEditor()

Suitable for string traits

Default for (none)

TitleEditor() generates a read-only display of a string value, formatted as a heading. All styles of the editor are identical. Visually, it is similar to a Heading item, but because it is an editor, you can change the text of the heading by modifying the underlying attribute.

TupleEditor()

Suitable for Tuple

Default for Tuple

Optional parameters *cols*, *editors*, *labels*, *traits*

The simple and custom editors generated by TupleEditor() provide a widget for each slot of the tuple being edited, based on the type of data in the slot. The text and read-only editors edit or display the text representation of the tuple.

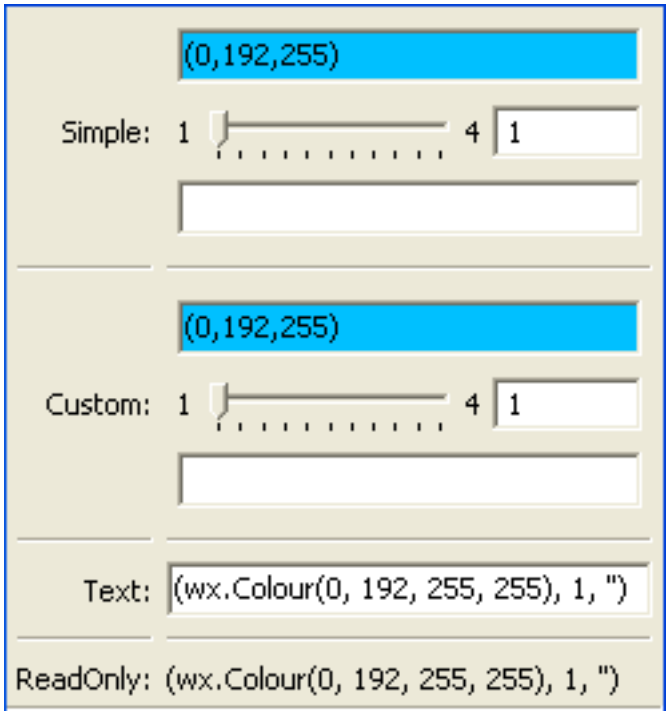


Fig. 1.48: Figure 48: Tuple editor styles

You can specify the number of columns to use to lay out the widgets with the *cols* parameter. You can specify labels for the widgets with the *labels* parameter. You can also specify trait definitions for the slots of the tuple; however, this is usually implicit in the tuple being edited.

You can supply a list of editors to be used for each corresponding tuple slot. If the *editors* list is missing, or is shorter than the length of the tuple, default editors are used for any tuple slots not defined in the list. This feature allows you to substitute editors, or to supply non-default parameters for editors.

ValueEditor()

Suitable for (any)

Default for (none)

Optional parameters *auto_open*

ValueEditor() generates a tree editor that displays Python values and objects, including all the objects' members. For example, Figure 49 shows a value editor that is displayed by the "pickle viewer" utility in enthought.debug.

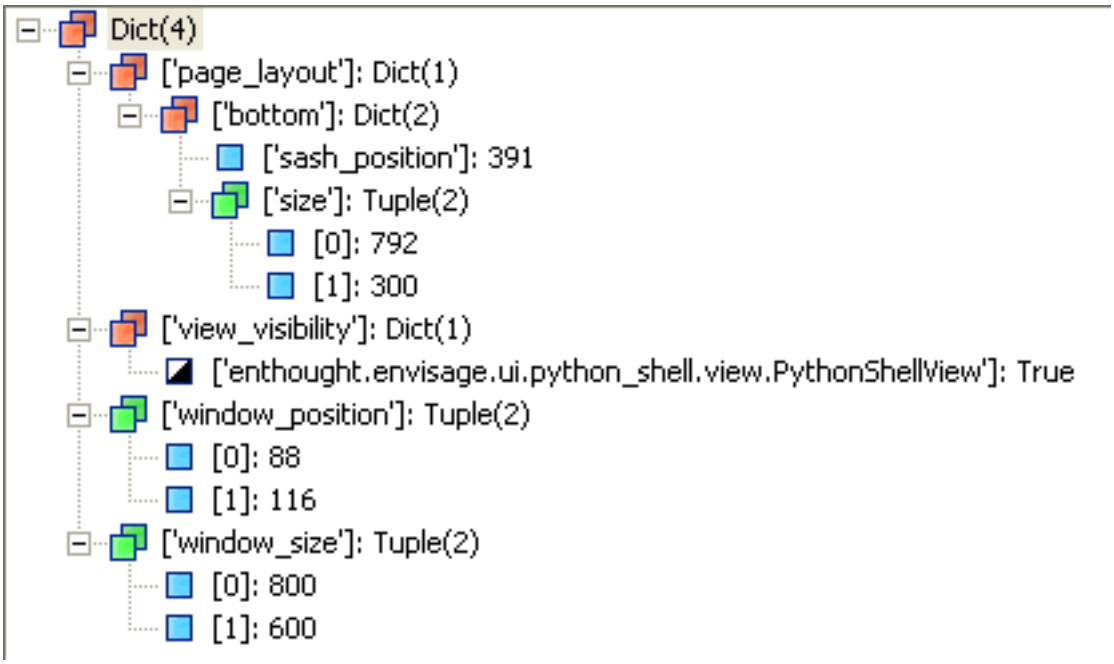


Fig. 1.49: Figure 49: Value editor from Pickle Viewer

1.10 Advanced Trait Editors

The editor factories described in the following sections are more advanced than those in the previous section. In some cases, they require writing additional code; in others, the editors they generate are intended for use in complex user interfaces, in conjunction with other editors.

1.10.1 CustomEditor()

Suitable for Special cases

Default for (none)

Required parameters *factory*

Optional parameters *args*

Use CustomEditor() to create an "editor" that is a non-Traits-based custom control. The *factory* parameter must be a function that generates the custom control. The function must have the following signature:

factory_function(*window_parent*, *editor*[, **args*, ***kwargs*])

- *window_parent*: The parent window for the control
- *editor*: The editor object created by CustomEditor()

Additional arguments, if any, can be passed as a tuple in the *args* parameter of CustomEditor().

For an example of using CustomEditor(),examine the implementation of the NumericModelExplorer class in the enthought.model.numeric_model_explorer module; CustomEditor() is used to generate the plots in the user interface.

1.10.2 DropEditor()

Suitable for Instance traits

Default for (none)

Optional parameters *binding*, *klass*, *readonly*

DropEditor() generates an editor that is a text field containing a string representation of the trait attribute's value. The user can change the value assigned to the attribute by dragging and dropping an object on the text field, for example, a node from a tree editor (See *TreeEditor()*). If the *readonly* parameter is True (the default), the user cannot modify the value by typing in the text field.

You can restrict the class of objects that can be dropped on the editor by specifying the *klass* parameter.

You can specify that the dropped object must be a binding (enthought.naming.api.Binding) by setting the *binding* parameter to True. If so, the bound object is retrieved and checked to see if it can be assigned to the trait attribute.

If the dropped object (or the bound object associated with it) has a method named drop_editor_value(), it is called to obtain the value to assign to the trait attribute. Similarly, if the object has a method named drop_editor_update(), it is called to update the value displayed in the text editor. This method requires one parameter, which is the GUI control for the text editor.

1.10.3 DNDEditor()

Suitable for Instance traits

Default for (none)

Optional parameters *drag_target, drop_target, image*

DNDEditor() generates an editor that represents a file or a HasTraits instance as an image that supports dragging and dropping. Depending on the editor style, the editor can be a *drag source* (the user can set the value of the trait attribute by dragging a file or object onto the editor, for example, from a tree editor), or *drop target* (the user can drag from the editor onto another target).

Table 9: Drag-and-drop editor style variations

Editor Style	Drag Source?	Drop Target?
Simple	Yes	Yes
Custom	No	Yes
Read-only	Yes	No

1.10.4 KeyBindingEditor()

The KeyBindingEditor() factory differs from other trait editor factories because it generates an editor, not for a single attribute, but for an object of a particular class, traitsui.key_bindings.KeyBindings. A KeyBindings object is a list of bindings between key codes and handler methods. You can specify a KeyBindings object as an attribute of a View. When the user presses a key while a View has input focus, the user interface searches the View for a KeyBindings that contains a binding that corresponds to the key press; if such a binding does not exist on the View, it searches enclosing Views in order, and uses the first matching binding, if any. If it does not find any matching bindings, it ignores the key press.

A key binding editor is a separate *dialog box* that displays the string representation of each key code and a description of the corresponding method. The user can click a text box, and then press a key or key combination to associate that key press with a method.

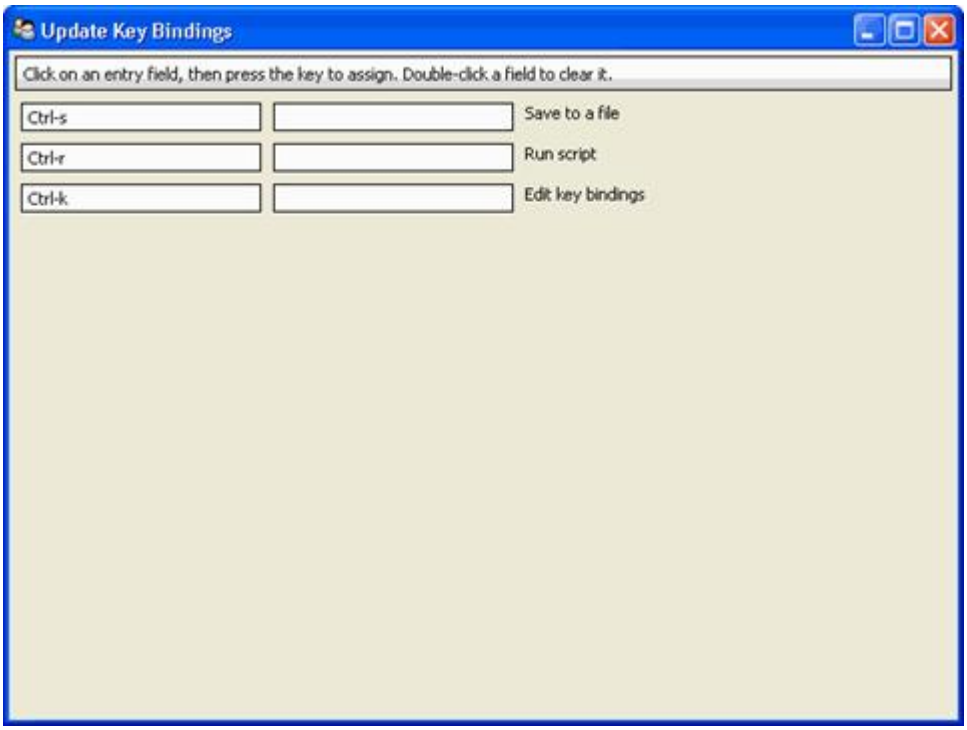

Update Key Bindings
Click on an entry field, then press the key to assign. Double-click a field to clear it.
Ctrl-s
Save to a file
Ctrl-r
Run script
Ctrl-k
Edit key bindings

Fig. 1.50: Figure 50: Key binding editor dialog box

The following code example creates a user interface containing a code editor with associated key bindings, and a button that invokes the key binding editor.

Example 17: Code editor with key binding editor

```
# key_bindings.py -- Example of a code editor with a
#                    key bindings editor

from traits.api \
    import Button, Code, HasPrivateTraits, Str
from traitsui.api \
    import View, Item, Group, Handler, CodeEditor
from traitsui.key_bindings \
    import KeyBinding, KeyBindings

key_bindings = KeyBindings(
    KeyBinding( binding1    = 'Ctrl-s',
                description = 'Save to a file',
                method_name = 'save_file' ),
    KeyBinding( binding1    = 'Ctrl-r',
                description = 'Run script',
                method_name = 'run_script' ),
    KeyBinding( binding1    = 'Ctrl-k',
                description = 'Edit key bindings',
                method_name = 'edit_bindings' )
)

# TraitsUI Handler class for bound methods
class CodeHandler ( Handler ):

    def save_file ( self, info ):
        info.object.status = "save file"

    def run_script ( self, info ):
        info.object.status = "run script"

    def edit_bindings ( self, info ):
        info.object.status = "edit bindings"
        key_bindings.edit_traits()

class KBCodeExample ( HasPrivateTraits ):

    code   = Code
    status = Str
    kb     = Button(label='Edit Key Bindings')

    view = View( Group (
                  Item( 'code',
                        style     = 'custom',
                        resizable = True ),
                  Item('status', style='readonly'),
                  'kb',
                  orientation = 'vertical',
                  show_labels = False,
                  ),
                id = 'KBCodeExample',
                key_bindings = key_bindings,
                title = 'Code Editor With Key Bindings',
                resizable = True,

                handler   = CodeHandler() )
```

```
    def _kb_fired( self, event ):
        key_bindings.edit_traits()


if __name__ == '__main__':
    KBCodeExample().configure_traits()
```

1.10.5 TableEditor()

Suitable for List(*InstanceType*)

Default for (none)

Required parameters *columns* or *columns_name*

Optional parameters See *Traits API Reference*, traitsui.wx.table_editor.ToolkitEditorFactory attributes.

TableEditor() generates an editor that displays instances in a list as rows in a table, with attributes of the instances as values in columns. You must specify the columns in the table. Optionally, you can provide filters for filtering the set of displayed items, and you can specify a wide variety of options for interacting with and formatting the table.

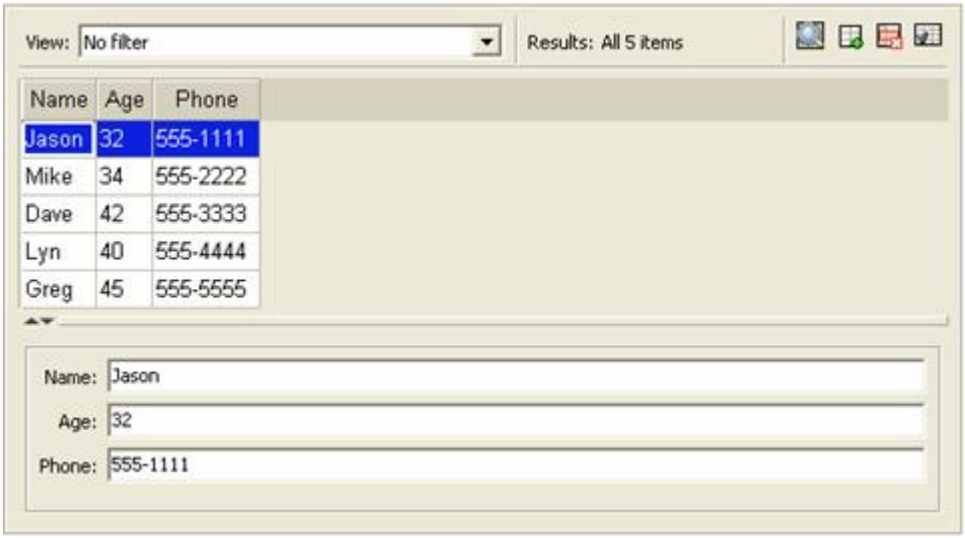


Fig. 1.51: Figure 51: Table editor

To see the code that results in Figure 51, refer to `TableEditor_demo.py` in the `demos/TraitsUI Demo/Standard Editors` subdirectory of the Traits UI package. This example demonstrates object columns, expression columns, filters, searching, and adding and deleting rows.

The parameters for TableEditor() can be grouped in several broad categories, described in the following sections.

- *Specifying Columns*
- *Managing Items*
- *Editing the Table*
- *Defining the Layout*
- *Defining the Format*
- *Other User Interactions*

Specifying Columns

You must provide the TableEditor() factory with a list of columns for the table. You can specify this list directly, as the value of the *columns* parameter, or indirectly, in an extended context attribute referenced by the *columns_name* parameter.

The items in the list must be instances of traitsui.api.TableColumn, or of a subclass of TableColumn. Some subclasses of TableColumn that are provided by the TraitsUI package include ObjectColumn, ListColumn, NumericColumn, and ExpressionColumn. (See the *Traits API Reference* for details about these classes.) In practice, most columns are derived from one of these subclasses, rather than from TableColumn. For the usual case of editing trait attributes on objects in the list, use ObjectColumn. You must specify the *name* parameter to the ObjectColumn() constructor, referencing the name of the trait attribute to be edited.

You can specify additional columns that are not initially displayed using the *other_columns* parameter. If the *configurable* parameter is True (the default), a *Set user preferences for table* icon ( ) appears on the table's toolbar. When the user clicks this icon, a dialog box opens that enables the user to select and order the columns displayed in the table, as shown in Figure 52. (The dialog box is implemented using a set editor; see *SetEditor()*.) Any columns that were specified in the *other_columns* parameter are listed in the left list box of this dialog box, and can be displayed by moving them into the right list box.

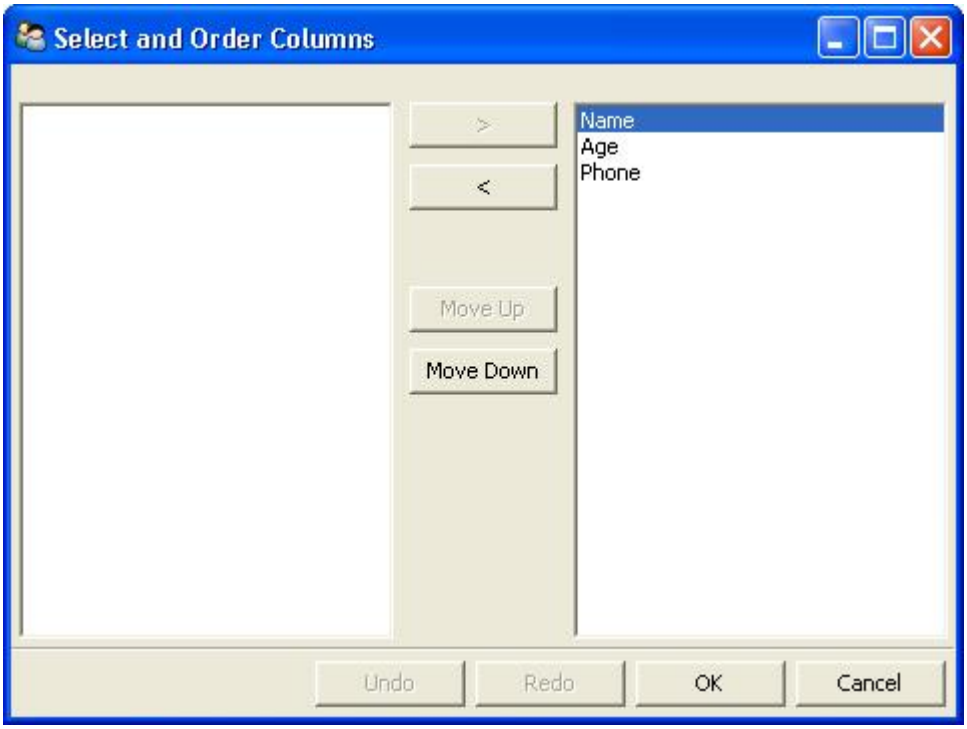


Fig. 1.52: Figure 52: Column selection dialog box for a table editor

Managing Items

Table editors support several mechanisms to help users locate items of interest.

Organizing Items

Table editors provide two mechanisms for the user to organize the contents of a table: sorting and reordering. The user can sort the items based on the values in a column, or the user can manually order the items. Usually, only one of these mechanisms is used in any particular table, although the TraitsUI package does not enforce a separation. If the user has manually ordered the items, sorting them would throw away that effort.

If the *reorderable* parameter is True, *Move up* ( ) and *Move down* ( ) icons appear in the table toolbar. Clicking one of these icons changes the position of the selected item.

If the *sortable* parameter is True (the default), then the user can sort the items in the table based on the values in a column by Control-clicking the header of that column.

- On the first click, the items are sorted in ascending order. The characters >> appear in the column header to indicate that the table is sorted ascending on this column's values.
- On the second click, the items are sorted descending order. The characters << appear in the column header to indicate that the table is sorted descending on this column's values.
- On the third click, the items are restored to their original order, and the column header is undecorated.

If the *sort_model* parameter is true, the items in the list being edited are sorted when the table is sorted. The default value is False, in which case, the list order is not affected by sorting the table.

If *sortable* is True and *sort_model* is False, then a *Do not sort columns* icon ( ) appears in the table toolbar. Clicking this icon restores the original sort order.

If the *reverse* parameter is True, then the items in the underlying list are maintained in the reverse order of the items in the table (regardless of whether the table is sortable or reorderable).

Filtering and Searching

You can provide an option for the user to apply a filter to a table, so that only items that pass the filter are displayed. This feature can be very useful when dealing with lengthy lists. You can specify a filter to apply to the table either directly, or via another trait. Table filters must be instances of traitsui.api.TableFilter, or of a subclass of TableFilter. Some subclasses of TableFilter that are provided by the TraitsUI package include EvalTableFilter, RuleTableFilter, and MenuTableFilter. (See the *Traits API Reference* for details about these classes.) The TraitsUI package also provides instances of these filter classes as "templates", which cannot be edited or deleted, but which can be used as models for creating new filters.

The *filter* parameter specifies a filter that is applied to the table when it is first displayed. The *filter_name* parameter specifies an extended trait name for a trait that is either a table filter object or a callable that accepts an object and returns True if the object passes the filter criteria, or false if it does not. You can use *filter_name* to embed a view of a table filter in the same view as its table.

You can specify use the *filters* parameter to specify a list of table filters that are available to apply to a table. When *filters* is specified, a drop-down list box appears in the table toolbar, containing the filters that are available for the user to apply. When the user selects a filter, it is automatically applied to the table. A status message to the right of the filters list indicates what subset of the items in the table is currently displayed. A special item in the filter list, named *Customize*, is always provided; clicking this item opens a dialog box that enables the user to create new filters, or to edit or delete existing filters (except templates).

You can also provide an option for the user to use filters to search the table. If you set the *search* parameter to an instance of TableFilter (or of a subclass), a *Search table* icon ( ) appears on the table toolbar. Clicking this icon

opens a *Search for* dialog box, which enables the user to specify filter criteria, to browse through matching items, or select all matching items.

Interacting with Items

As the user clicks in the table, you may wish to enable certain program behavior.

The value of the *selection_mode* parameter specifies how the user can make selections in the grid:

- `cell`: A single cell at a time
- `cells`: Multiple cells
- `column`: A single column at a time
- `columns`: Multiple columns
- `row`: A single row at a time
- `rows`: Multiple rows

You can use the *selected* parameter to specify the name of a trait attribute in the current context to synchronize with the user's current selection. For example, you can enable or disable menu items or toolbar icons depending on which item is selected. The synchronization is two-way; you can set the attribute referenced by *selected* to force the table to select a particular item.

You can use the *selected_indices* parameter to specify the name of a trait attribute in the current context to synchronize with the indices of the table editor selection. The content of the selection depends on the *selection_mode* value:

- **`cell`: The selection is a tuple of the form (*object*, *column_name*),** where *object* is the object contains the selected cell, and *column_name* is the name of the column the cell is in. If there is no selection, the tuple is (None, '').
- `cells`: The selection is a list of tuples of the form (*object*, *column_name*), with one tuple for each selected cell, in order from top to bottom and left to right. If there is no selection, the list is empty.
- `column`: The selection is the name of the selected column, or the empty string if there is no selection.
- `columns`: The selection is a list containing the names of the selected columns, in order from left to right. If there is no selection, the list is empty.
- `row`: The selection is either the selected object or None if nothing is selected in the table.
- `rows`: The selection is a list of the selected objects, in ascending row order. If there is no selection, the list is empty.

The *on_select* and *on_dclick* parameters are callables to invoke when the user selects or double-clicks an item, respectively.

You can define a shortcut menu that opens when the user right-clicks an item. Use the *menu* parameter to specify a TraitsUI or PyFace Menu, containing Action objects for the menu commands.

Editing the Table

The Boolean *editable* parameter controls whether the table or its items can be modified in any way. This parameter defaults to True, except when the style is 'readonly'. Even when the table as a whole is editable, you can control whether individual columns are editable through the **editable** attribute of TableColumn.

Adding Items

To enable users to add items to the table, specify as the *row_factory* parameter a callable that generates an object that can be added to the list in the table; for example, the class of the objects in the table. When *row_factory* is specified, an *Insert new item* icon ( ) appears in the table toolbar, which generates a new row in the table. Optionally, you can use *row_factory_args* and *row_factory_kw* to specify positional and keyword arguments to the row factory callable.

To save users the trouble of mousing to the toolbar, you can enable them to add an item by selecting the last row in the table. To do this, set *auto_add* to True. In this case, the last row is blank until the user sets values. Pressing Enter creates the new item and generates a new, blank last row.

Deleting Items

The *deletable* parameter controls whether items can be deleted from the table. This parameter can be a Boolean (defaulting to False) or a callable; the callable must take an item as an argument and handle deleting it. If *deletable* is not False, a *Delete current item* icon ( ) appears on the table toolbar; clicking it deletes the item corresponding to the row that is selected in the table.

Modifying Items

The user can modify items in two ways.

- For columns that are editable, the user can change an item's value directly in the table. The editor used for each attribute in the table is the simple style of editor for the corresponding trait.
- Alternatively, you can specify a View for editing instances, using the *edit_view* parameter. The resulting user interface appears in a *subpanel* to the right or below the table (depending on the *orientation* parameter). You can specify a handler to use with the view, using *edit_view_handler*. You can also specify the subpanel's height and width, with *edit_view_height* and *edit_view_width*.

Defining the Layout

Some of the parameters for the TableEditor() factory affect global aspects of the display of the table.

- *auto_size*: If True, the cells of the table automatically adjust to the optimal size based on their contents.
- *orientation*: The layout of the table relative to its associated editor pane. Can be 'horizontal' or 'vertical'.
- *rows*: The number of visible rows in the table.
- *show_column_labels*: If True (the default), displays labels for the columns. You can specify the labels to use in the column definitions; otherwise, a "user friendly" version of the trait attribute name is used.
- *show_toolbar*: If False, the table toolbar is not displayed, regardless of whether other settings would normally create a toolbar. The default is True.

Defining the Format

The TableEditor() factory supports a variety of parameters to control the visual formatting of the table, such as colors, fonts, and sizes for lines, cells, and labels. For details, refer to the *Traits API Reference*, traitsui.wx.table_editor.ToolkitEditorFactory attributes.

You can also specify formatting options for individual table columns when you define them.

Other User Interactions

The table editor supports additional types of user interaction besides those controlled by the factory parameters.

- Column dragging: The user can reorganize the column layout of a table editor by clicking and dragging a column label to its new location. If you have enabled user preferences for the view and table editor (by specifying view and item IDs), the new column layout is persisted across user sessions.
- Column resizing: The user can resize a column by dragging the column separator (in one of the data rows) to a new position. Because of the column-dragging support, clicking the column separator in the column label row does not work.
- Data dragging: The user can drag the contents of any cell by clicking and dragging.

1.10.6 TabularEditor()

Suitable for lists, arrays, and other large sequences of objects

Default for (none)

Required parameters *adapter*

Optional parameters *activated, clicked, column_clicked, dclicked, drag_move, editable, horizontal_lines, images, multi_select, operations, right_clicked, right_dclicked, selected, selected_row, show_titles, vertical_lines*

The TabularEditor() factory can be used for many of the same purposes as the TableEditor() factory, that is, for displaying a table of attributes of lists or arrays of objects. While similar in function, the tabular editor has advantages and disadvantages relative to the table editor.

Advantages

- **Very fast**: The tabular editor uses a virtual model, which accesses data from the underlying model only as needed. For example, if you have a million-element array, but can display only 50 rows at a time, the editor requests only 50 elements of data at a time.
- **Very flexible data model**: The editor uses an adapter model to interface with the underlying data. This strategy allows it to easily deal with many types of data representation, from list of objects, to arrays of numbers, to tuples of tuples, and many other formats.
- **Supports useful data operations**, including:
  - Moving the selection up and down using the keyboard arrow keys.
  - Moving rows up and down using the keyboard.
  - Inserting and deleting items using the keyboard.
  - Initiating editing of items using the keyboard.
  - Dragging and dropping of table items to and from the editor, including support for both copy and move operations for single and multiple items.
- **Visually appealing**: The tabular editor, in general, uses the underlying operating system's native table or grid control, and as a result often looks better than the control used by the table editor.
- **Supports displaying text and images in any cell**. However, the images displayed must be all the same size for optimal results.

Disadvantages

- **Not as full-featured**: The table editor includes support for arbitrary data filters, searches, and different types of sorting. These differences may narrow as features are added to the tabular editor.
- **Limited data editing capabilities**: The tabular editor supports editing only textual values, whereas the table editor supports a wide variety of column editors, and can be extended with more as needed. This is due to limitations of the underlying native control used by the tabular editor.

TabularAdapter

The tabular editor works in conjunction with an adapter class, derived from TabularAdapter. The tabular adapter interfaces between the tabular editor and the data being displayed. The tabular adapter is the reason for the flexibility and power of the tabular editor to display a wide variety of data.

The most important attribute of TabularAdapter is **columns**, which is list of columns to be displayed. Each entry in the **columns** list can be either a string, or a tuple consisting of a string and another value, which can be of any type. The string is used as the label for the column. The second value in the tuple, called the *column ID*, identifies the column to the adapter. It is typically a trait attribute name or an integer index, but it can be any value appropriate to the adapter. If only a string is specified for an entry, then the index of the entry within the **columns** list is used as that entry's column ID.

Attributes on TabularAdapter control the appearance of items, and aspects of interaction with items, such as whether they can be edited, and how they respond to dragging and dropping. Setting any of these attributes on the adapter subclass sets the global behavior for the editor. Refer to the *Traits API Reference* for details of the available attributes.

You can also specify these attributes for a specific class or column ID, or combination of class and column ID. When the TabularAdapter needs to look up the value of one of its attributes for a specific item in the table, it looks for attributes with the following naming conventions in the following order:

1. *classname_columnid_attribute*
2. *classname_attribute*
3. *columnid_attribute*
4. *attribute*

For example, to find the **text_color** value for an item whose class is Person and whose column ID is 'age', the get_text_color() method looks for the following attributes in sequence, and returns the first value it finds:

1. **Person_age_text_color**
2. **Person_text_color**
3. **age_text_color**
4. **text_color**

Note that the *classname* can be the name of a base class, searched in the method resolution order (MRO) for the item's class. So for example, if the item were a direct instance of Employee, which is a subclass of Person, then the **Person_age_text_color** attribute would apply to that item (as long as there were no **Employee_age_text_color** attribute).

The Tabular Editor User Interface

Figure 53 shows an example of a tabular editor on Microsoft Windows, displaying information about source files in the Traits package. This example includes a column that contains an image for files that meet certain conditions.

File Name	Size	Time	Date
adapter.py	7830	04:08:06 PM	08/13/2007
api.py	5794	05:37:52 PM	11/16/2007
category.py	4757	04:08:06 PM	08/13/2007
core_traits.py	3758	04:08:06 PM	08/13/2007
has_dynamic_views.py	15556	04:08:06 PM	08/13/2007
has_traits.py	146863	03:29:33 PM	11/28/2007
standard.py	13715	04:08:06 PM	08/13/2007
traits.py	55643	05:37:52 PM	11/16/2007
traits_listener.py	39379	03:29:33 PM	11/28/2007
trait_base.py	15628	12:10:01 PM	11/09/2007
trait_db.py	22879	04:08:06 PM	08/13/2007
trait_errors.py	4058	12:11:35 PM	09/11/2007
trait_handlers.py	112372	12:10:01 PM	11/09/2007
trait_notifiers.py	27635	01:28:49 PM	10/03/2007
trait_numeric.py	14070	12:10:01 PM	11/09/2007

Fig. 1.53: Figure 53: Tabular editor on MS Windows

Depending on how the tabular editor is configured, certain keyboard interactions may be available. For some interactions, you must specify that the corresponding operation is allowed by including the operation name in the *operations* list parameter of TabularEditor().

- `Up arrow`: Selects the row above the currently selected row.
- `Down arrow`: Selects the row below the currently selected row.
- `Page down`: Appends a new item to the end of the list ('append' operation).
- `Left arrow`: Moves the currently selected row up one line ('move' operation).
- `Right arrow`: Moves the currently selected row down one line ('move' operation).
- `Backspace, Delete`: Deletes from the list all items in the current selection ('delete' operation).
- `Enter, Escape`: Initiates editing on the current selection ('edit' operation).
- **`Insert`: Inserts a new item before the current selection ('insert'** operation).

The 'append', 'move', 'edit', and 'insert' operations can occur only when a single item is selected. The 'delete' operation works for one or more items selected.

Depending on how the editor and adapter are specified, drag and drop operations may be available. If the user selects multiple items and drags one of them, all selected items are included in the drag operation. If the user drags a non-selected item, only that item is dragged.

The editor supports both "drag-move" and "drag-copy" semantics. A drag-move operation means that the dragged items are sent to the target and are removed from the list displayed in the editor. A drag-copy operation means that the dragged items are sent to the target, but are not deleted from the list data.

1.10.7 TreeEditor()

Suitable for Instance

Default for (none)

Required parameters *nodes* (required except for shared editors; see *Editing Objects*)

Optional parameters *auto_open, editable, editor, hide_root, icon_size, lines_mode, on_dclick, on_select, orientation, selected, shared_editor, show_icons*

TreeEditor() generates a hierarchical tree control, consisting of nodes. It is useful for cases where objects contain lists of other objects.

The tree control is displayed in one pane of the editor, and a user interface for the selected object is displayed in the other pane. The layout orientation of the tree and the object editor is determined by the *orientation* parameter of TreeEditor(), which can be 'horizontal' or 'vertical'.

You must specify the types of nodes that can appear in the tree using the *nodes* parameter, which must be a list of instances of TreeNode (or of subclasses of TreeNode).

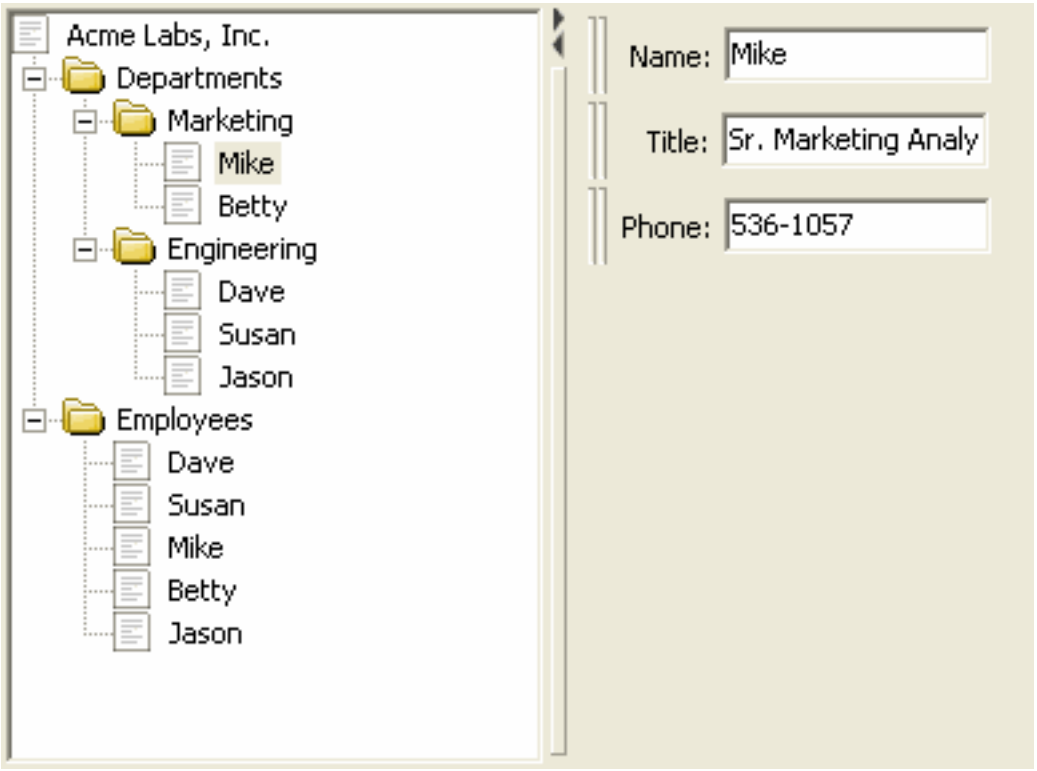


Fig. 1.54: Figure 54: Tree editor

The following example shows the code that produces the editor shown in Figure 54.

Example 18: Code for example tree editor

```
# tree_editor.py -- Example of a tree editor

from traits.api \
    import HasTraits, Str, Regex, List, Instance
from traitsui.api \
    import TreeEditor, TreeNode, View, Item, VSplit, \
           HGroup, Handler, Group
from traitsui.menu \
    import Menu, Action, Separator
from traitsui.wx.tree_editor \
    import NewAction, CopyAction, CutAction, \
           PasteAction, DeleteAction, RenameAction
```

```
# DATA CLASSES

class Employee ( HasTraits ):
    name  = Str( '<unknown>' )
    title = Str
    phone = Regex( regex = r'\d\d\d-\d\d\d\d' )

    def default_title ( self ):
        self.title = 'Senior Engineer'

class Department ( HasTraits ):
    name      = Str( '<unknown>' )
    employees = List( Employee )


class Company ( HasTraits ):
    name        = Str( '<unknown>' )
    departments = List( Department )
    employees   = List( Employee )

class Owner ( HasTraits ):
    name    = Str( '<unknown>' )
    company = Instance( Company )

# INSTANCES

jason = Employee(
     name  = 'Jason',
     title = 'Engineer',
     phone = '536-1057' )

mike = Employee(
     name  = 'Mike',
     title = 'Sr. Marketing Analyst',
     phone = '536-1057' )

dave = Employee(
     name  = 'Dave',
     title = 'Sr. Engineer',
     phone = '536-1057' )

susan = Employee(
     name  = 'Susan',
     title = 'Engineer',
     phone = '536-1057' )

betty = Employee(
     name  = 'Betty',
     title = 'Marketing Analyst' )

owner = Owner(
    name    = 'wile',
    company = Company(
        name = 'Acme Labs, Inc.',
        departments = [
            Department(
                name = 'Marketing',
                employees = [ mike, betty ]
```

```
            ),
            Department(
                name = 'Engineering',
                employees = [ dave, susan, jason ]
            )
        ],
        employees = [ dave, susan, mike, betty, jason ]
    )
)

# View for objects that aren't edited
no_view = View()

# Actions used by tree editor context menu

def_title_action = Action(name='Default title',
                          action = 'object.default')

dept_action = Action(
    name='Department',
    action='handler.employee_department(editor,object)')

# View used by tree editor
employee_view = View(
    VSplit(
        HGroup( '3', 'name' ),
        HGroup( '9', 'title' ),
        HGroup( 'phone' ),
        id = 'vsplit' ),
    id = 'traits.doc.example.treeeditor',
    dock = 'vertical' )

class TreeHandler ( Handler ):

    def employee_department ( self, editor, object ):
        dept = editor.get_parent( object )
        print '%s works in the %s department.' %\
            ( object.name, dept.name )

# Tree editor
tree_editor = TreeEditor(
    nodes = [
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = '',
                  label     = 'name',
                  view      = View( Group('name',
                                          orientation='vertical',
                                          show_left=True )) ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = 'departments',
                  label     = '=Departments',
                  view      = no_view,
                  add       = [ Department ] ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = 'employees',
```

```
                    label     = '=Employees',
                    view      = no_view,
                    add       = [ Employee ] ),
            TreeNode( node_for  = [ Department ],
                    auto_open = True,
                    children  = 'employees',
                    label     = 'name',
                    menu      = Menu( NewAction,
                                      Separator(),
                                      DeleteAction,
                                      Separator(),
                                      RenameAction,
                                      Separator(),
                                      CopyAction,
                                      CutAction,
                                      PasteAction ),
                    view      = View( Group ('name',
                                     orientation='vertical',
                                     show_left=True )),
                    add       = [ Employee ] ),
            TreeNode( node_for  = [ Employee ],
                    auto_open = True,
                    label     = 'name',
                    menu=Menu( NewAction,
                               Separator(),
                               def_title_action,
                               dept_action,
                               Separator(),
                               CopyAction,
                               CutAction,
                               PasteAction,
                               Separator(),
                               DeleteAction,
                               Separator(),
                               RenameAction ),
                    view = employee_view )
        ]
)

# The main view
view = View(
            Group(
                Item(
                     name = 'company',
                     id = 'company',
                     editor = tree_editor,
                     resizable = True ),
                 orientation = 'vertical',
                 show_labels = True,
                 show_left = True, ),
             title = 'Company Structure',
             id = \
              'traitsui.tests.tree_editor_test',
             dock = 'horizontal',
             drop_class = HasTraits,
             handler = TreeHandler(),
             buttons = [ 'Undo', 'OK', 'Cancel' ],
             resizable = True,
```

```
            width = .3,
            height = .3 )

if __name__ == '__main__':
    owner.configure_traits( view = view )
```

Defining Nodes

For details on the attributes of the TreeNode class, refer to the *Traits API Reference*.

You must specify the classes whose instances the node type applies to. Use the **node_for** attribute of TreeNode to specify a list of classes; often, this list contains only one class. You can have more than one node type that applies to a particular class; in this case, each object of that class is represented by multiple nodes, one for each applicable node type. In Figure 54, one Company object is represented by the nodes labeled "Acme Labs, Inc.", "Departments", and "Employees".

A Node Type without Children

To define a node type without children, set the **children** attribute of TreeNode to the empty string. In Example 16, the following lines define the node type for the node that displays the company name, with no children:

```
TreeNode( node_for  = [ Company ],
          auto_open = True,
          children  = '',
          label     = 'name',
          view      = View( Group('name',
                            orientation='vertical',
                            show_left=True )) ),
```

A Node Type with Children

To define a node type that has children, set the **children** attribute of TreeNode to the (extended) name of a trait on the object that it is a node for; the named trait contains a list of the node's children. In Example 16, the following lines define the node type for the node that contains the departments of a company. The node type is for instances of Company, and 'departments' is a trait attribute of Company.

```
TreeNode( node_for  = [ Company ],
          auto_open = True,
          children  = 'departments',
          label     = '=Departments',
          view      = no_view,
          add       = [ Department ] ),
```

Setting the Label of a Tree Node

The **label** attribute of Tree Node can work in either of two ways: as a trait attribute name, or as a literal string.

If the value is a simple string, it is interpreted as the extended trait name of an attribute on the object that the node is for, whose value is used as the label. This approach is used in the code snippet in *A Node Type without Children*.

If the value is a string that begins with an equals sign ('='), the rest of the string is used as the literal label. This approach is used in the code snippet in *A Node Type with Children*.

You can also specify a callable to format the label of the node, using the **formatter** attribute of TreeNode.

Defining Operations on Nodes

You can use various attributes of TreeNode to define operations or behavior of nodes.

Shortcut Menus on Nodes

Use the **menu** attribute of TreeNode to define a shortcut menu that opens when the user right-clicks on a node. The value is a TraitsUI or PyFace menu containing Action objects for the menu commands. In Example 16, the following lines define the node type for employees, including a shortcut menu for employee nodes:

```
TreeNode( node_for  = [ Department ],
          auto_open = True,
          children  = 'employees',
          label     = 'name',
          menu      = Menu( NewAction,
                            Separator(),
                            DeleteAction,
                            Separator(),
                            RenameAction,
                            Separator(),
                            CopyAction,
                            CutAction,
                            PasteAction ),
          view      = View( Group ('name',
                                   orientation='vertical',
                                   show_left=True )),
          add       = [ Employee ] ),
```

Allowing the Hierarchy to Be Modified

If a node contains children, you can allow objects to be added to its set of children, through operations such as dragging and dropping, copying and pasting, or creating new objects. Two attributes control these operations: **add** and **move**. Both are lists of classes. The **add** attribute contains classes that can be added by any means, including creation. The code snippet in the preceding section includes an example of the **add** attribute. The **move** attribute contains classes that can be dragged and dropped, but not created. The **move** attribute need not be specified if all classes that can be moved can also be created (and therefore are specified in the **add** value).

Note: The **add** attribute alone is not enough to create objects.

Specifying the **add** attribute makes it possible for objects of the specified classes to be created, but by itself, it does not provide a way for the user to do so. In the code snippet in the preceding section (*Shortcut Menus on Nodes*), 'NewAction' in the Menu constructor call defines a *New* > *Employee* menu item that creates Employee objects.

In the example tree editor, users can create new employees using the *New* > *Employee* shortcut menu item, and they can drag an employee node and drop it on a department node. The corresponding object becomes a member of the appropriate list.

You can specify the label that appears on the *New* submenu when adding a particular type of object, using the **name** attribute of TreeNode. Note that you set this attribute on the tree node type that will be *added* by the menu item, not the node type that *contains* the menu item. For example, to change *New* > *Employee* to *New* > *Worker*, set `name = 'Worker'` on the tree node whose **node_for** value contains Employee. If this attribute is not set, the class name is used.

You can determine whether a node or its children can be copied, renamed, or deleted, by setting the following attributes on TreeNode:

Attribute	If True, the ...	can be...
copy	object's children	copied.
delete	object's children	deleted.
delete_me	object	deleted.
rename	object's children	renamed.
rename_me	object	renamed.

All of these attributes default to True. As with **add**, you must also define actions to perform these operations.

Behavior on Nodes

As the user clicks in the tree, you may wish to enable certain program behavior.

You can use the *selected* parameter to specify the name of a trait attribute on the current context object to synchronize with the user's current selection. For example, you can enable or disable menu items or toolbar icons depending on which node is selected. The synchronization is two-way; you can set the attribute referenced by *selected* to force the tree to select a particular node.

The *on_select* and *on_dclick* parameters are callables to invoke when the user selects or double-clicks a node, respectively.

Expanding and Collapsing Nodes

You can control some aspects of expanding and collapsing of nodes in the tree.

The integer *auto_open* parameter of TreeEditor() determines how many levels are expanded below the root node, when the tree is first displayed. For example, if *auto_open* is 2, then two levels below the root node are displayed (whether or not the root node itself is displayed, which is determined by *hide_root*).

The Boolean **auto_open** attribute of TreeNode determines whether nodes of that type are expanded when they are displayed (at any time, not just on initial display of the tree). For example, suppose that a tree editor has *auto_open* setting of 2, and contains a tree node at level 3 whose **auto_open** attribute is True. The nodes at level 3 are not displayed initially, but when the user expands a level 2 node, displaying the level 3 node, that's nodes children are automatically displayed also. Similarly, the number of levels of nodes initially displayed can be greater than specified by the tree editor's *auto_open* setting, if some of the nodes have **auto_open** set to True.

If the **auto_close** attribute of TreeNode is set to True, then when a node is expanded, any siblings of that node are automatically closed. In other words, only one node of this type can be expanded at a time.

Editing Objects

One pane of the tree editor displays a user interface for editing the object that is selected in the tree. You can specify a View to use for each node type using the **view** attribute of TreeNode. If you do not specify a view, then the default view for the object is displayed. To suppress the editor pane, set the *editable* parameter of TreeEditor() to False; in this case, the objects represented by the nodes can still be modified by other means, such as shortcut menu commands.

You can define multiple tree editors that share a single editor pane. Each tree editor has its own tree pane. Each time the user selects a different node in any of the sharing tree controls, the editor pane updates to display the user interface for the selected object. To establish this relationship, do the following:

1. Call TreeEditor() with the *shared_editor* parameter set to True, without defining any tree nodes. The object this call returns defines the shared editor pane. For example:

```
my_shared_editor_pane = TreeEditor(shared_editor=True)
```

2. For each editor that uses the shared editor pane:

- Set the *shared_editor* parameter of TreeEditor() to True.
- Set the editor parameter of TreeEditor() to the object returned in Step 1.

For example:

```
shared_tree_1 = TreeEditor(shared_editor = True,
                           editor = my_shared_editor_pane,
                           nodes = [ TreeNode( # ...
                                               )
                                   ]
                           )
shared_tree_2 = TreeEditor(shared_editor = True,
                           editor = my_shared_editor_pane,
                           nodes = [ TreeNode( # ...
                                               )
                                   ]
                           )
```

Defining the Format

Several parameters to TreeEditor() affect the formatting of the tree control:

- *show_icons*: If True (the default), icons are displayed for the nodes in the tree.
- *icon_size*: A two-integer tuple indicating the size of the icons for the nodes.
- *lines_mode*: Determines whether lines are displayed between related nodes. The valid values are 'on', 'off', and 'appearance' (the default). When set to 'appearance', lines are displayed except on Posix-based platforms.
- *hide_root*: If True, the root node in the hierarchy is not displayed. If this parameter were specified as True in Example 16, the node in Figure 54 that is labeled "Acme Labs, Inc." would not appear.

Additionally, several attributes of TreeNode also affect the display of the tree:

- **icon_path**: A directory path to search for icon files. This path can be relative to the module it is used in.
- **icon_item**: The icon for a leaf node.
- **icon_open**: The icon for a node with children whose children are displayed.
- **icon_group**: The icon for a node with children whose children are not displayed.

The wxWidgets implementation automatically detects the bitmap format of the icon.

1.11 "Extra" Trait Editor Factories

The traitsui.wx package defines a few editor factories that are specific to the wxWidgets toolkit, some of which are also specific to the Microsoft Windows platform. These editor factories are not necessarily implemented for other GUI toolkits or other operating system platforms.

1.11.1 AnimatedGIFEditor()

Suitable for File

Default for (none)

Optional parameters *playing*

AnimatedGIFEditor() generates a display of the contents of an animated GIF image file. The Boolean *playing* parameter determines whether the image is animated or static.

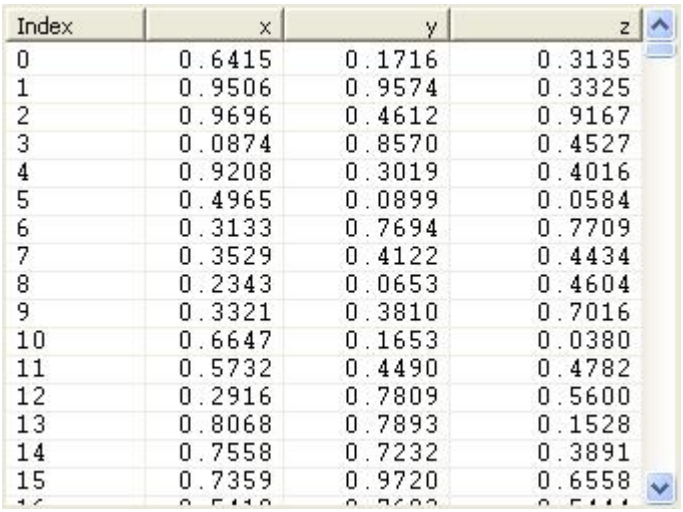
1.11.2 ArrayViewEditor()

Suitable for 2-D Array, 2-D CArray

Default for (none)

Optional parameters *format, show_index, titles, transpose*

ArrayViewEditor() generates a tabular display for an array. It is suitable for use with large arrays, which do not work well with the editors generated by ArrayEditor(). All styles of the editor have the same appearance.



Index	x	y	z
0	0.6415	0.1716	0.3135
1	0.9506	0.9574	0.3325
2	0.9696	0.4612	0.9167
3	0.0874	0.8570	0.4527
4	0.9208	0.3019	0.4016
5	0.4965	0.0899	0.0584
6	0.3133	0.7694	0.7709
7	0.3529	0.4122	0.4434
8	0.2343	0.0653	0.4604
9	0.3321	0.3810	0.7016
10	0.6647	0.1653	0.0380
11	0.5732	0.4490	0.4782
12	0.2916	0.7809	0.5600
13	0.8068	0.7893	0.1528
14	0.7558	0.7232	0.3891
15	0.7359	0.9720	0.6558

Fig. 1.55: Figure 55: Array view editor

1.11.3 DataFrameEditor()

Suitable for Pandas DataFrames

Default for (none)

Optional parameters *formats, show_index, show_titles, columns, fonts, selected, selected_row, selectable, activated, activated_row, clicked, dclicked, right_clicked, right_dclicked, column_clicked, column_right_clicked, editable, operations*

DataFrameEditor() generates a tabular display for a DataFrame. It is suitable for use with large DataFrames. All styles of the editor have the same appearance. Many of the optional parameters are identical to those of the TabularEditor().

The following have special meaning for the DataFrameEditor():

- **formats**: either a %-style formatting string for all entries, or a dictonary mapping DataFrame columns to formatting strings.
- **show_index**: whether or not to show the index as a column in the table.
- **show_titles**: whether or not to show column headers on the table.
- **fonts**: either a font for all entries, or a mapping of column id to fonts.

1.11.4 FlashEditor()

Suitable for string traits, Enum(string values)

Default for (none)

FlashEditor() generates a display of an Adobe Flash Video file, using an ActiveX control (if one is installed on the system). This factory is available only on Microsoft Windows platforms. The attribute being edited must have a value whose text representation is the name or URL of a Flash video file. If the value is a Unicode string, it must contain only characters that are valid for filenames or URLs.

1.11.5 HistoryEditor()

Suitable for string traits

Default for (none)

Optional parameters *entries*

HistoryEditor() generates a combo box, which allows the user to either enter a text string or select a value from a list of previously-entered values. The same control is used for all editor styles. The *entries* parameter determines how many entries are preserved in the history list. This type of control is used as part of the simple style of file editor; see *FileEditor()*.

1.11.6 IEHTMLEditor()

Suitable for string traits, Enum(string values)

Default for (none)

Optional parameters *back, forward, home, html, page_loaded, refresh, search, status, stop, title*

IEHTMLEditor() generates a display of a web page, using Microsoft Internet Explorer (IE) via ActiveX to render the page. This factory is available only on Microsoft Windows platforms. The attribute being edited must have value whose text representation is a URL. If the value is a Unicode string, it must contain only characters that are valid for URLs.

The *back*, *forward*, *home*, *refresh*, *search* and *stop* parameters are extended names of event attributes that represent the user clicking on the corresponding buttons in the standard IE interface. The IE buttons are not displayed by the editor; you must create buttons separately in the View, if you want the user to be able to actually click buttons.

The *html*, *page_loaded*, *status*, and *title* parameters are the extended names of string attributes, which the editor updates with values based on its own state. You can display these attributes elsewhere in the View.

- *html*: The current page content as HTML (as would be displayed by the *View* > *Source* command in IE).
- *page_loaded*: The URL of the currently displayed page; this may be different from the URL represented by the attribute being edited.
- *status*: The text that would appear in the IE status bar.
- *title*: The title of the currently displayed page.

1.11.7 ImageEditor()

Suitable for (any)

Default for (none)

Optional parameters *image*, *scale*, *preserve_aspect_ratio*, *allow_upscaling*, *allow_clipping*

ImageEditor() generates a read-only display of an image. The image to be displayed is determined by the *image* parameter, or by the value of the trait attribute being edited, if *image* is not specified. In either case, the value must be a PyFace ImageResource (pyface.api.ImageResource), or a string that can be converted to one. If *image* is specified, then the type and value of the trait attribute being edited are irrelevant and are ignored.

For the Qt backend *scale*, *preserve_aspect_ratio*, *allow_upscaling*, and *allow_clipping* control whether the image should be scaled or not, and how to perform that scaling.

1.11.8 LEDEditor()

Suitable for numeric traits

Default for (none)

Optional parameters *alignment, format_str*

LEDEditor() generates a display that resembles a "digital" display using light-emitting diodes. All styles of this editor are the same, and are read-only.

The *alignment* parameter can be 'left', 'center', or 'right' to indicate how the value should be aligned within the display. The default is right-alignment.

Fig. 1.56: Figure 56: LED Editor with right alignment

1.11.9 ThemedButtonEditor()

Suitable for Event

Default for (none)

Optional parameters *label*, *theme, down_theme, hover_theme, disabled_theme, image, position, spacing, view*

The ThemedButtonEditor() factory generates a button that is formatted according to specified or default themes. All editor styles have the same appearance.

The theme-related parameters determine the appearance of the button in various states. Figure 57 shows the default theme.

1.11.10 ThemedCheckboxEditor()

Suitable for Boolean

Default for (none)

Optional parameters *label*, *theme, hover_off_image, hover_off_theme, hover_on_image, hover_on_theme, image, on_image, on_theme, position, spacing*

Fig. 1.57: Figure 57: Themed buttons in various states

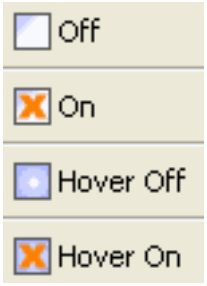


Fig. 1.58: Figure 58: Themed checkbox in various states

The ThemedCheckboxEditor() factory generates a checkbox that is formatted according to specified or default themes. All editor styles have the same appearance.

The theme-related parameters determine the appearance of the checkbox in the various states. shows the default theme. If *label* is not specified for the editor factory, the value is inherited from the *label* value of the enclosing Item. Both labels may be displayed, if the Item's label is not hidden.

1.11.11 ThemedSliderEditor()

Suitable for Range

Default for (none)

Optional parameters *alignment, bg_color, high, increment, low, show_value, slider_color, text_color, tip_color*

The ThemedSliderEditor() factory generates a slider control that is formatted according to specified or default themes. All editor styles have the same appearance. The value is edited by modifying its textual representation. The background of the control updates to reflect the value relative to the total range represented by a slider. For example, if the range is from -2 to 2, a value of 0 is represented by a bar covering the left half of the control area, as shown in Figure 59.

Fig. 1.59: Figure 59: Themed slider without focus, and with focus

1.11.12 ThemedTextEditor()

Suitable for Str, String, Unicode, CStr, CUnicode, and any trait whose value is a string

Default for (none)

Optional parameters *auto_set*, *enter_set*, *evaluate*, *evaluate_name*, *mapping*, *multi_line*, *password*, *theme*

The ThemedTextEditor() factory generates a text editor that is formatted according to a specified theme. If no theme is specified, the editor uses the theme, if any, specified by the surrounding Group or View. Thus, there is no default theme. All editor styles have the same appearance, except the read-only style, which is not editable.

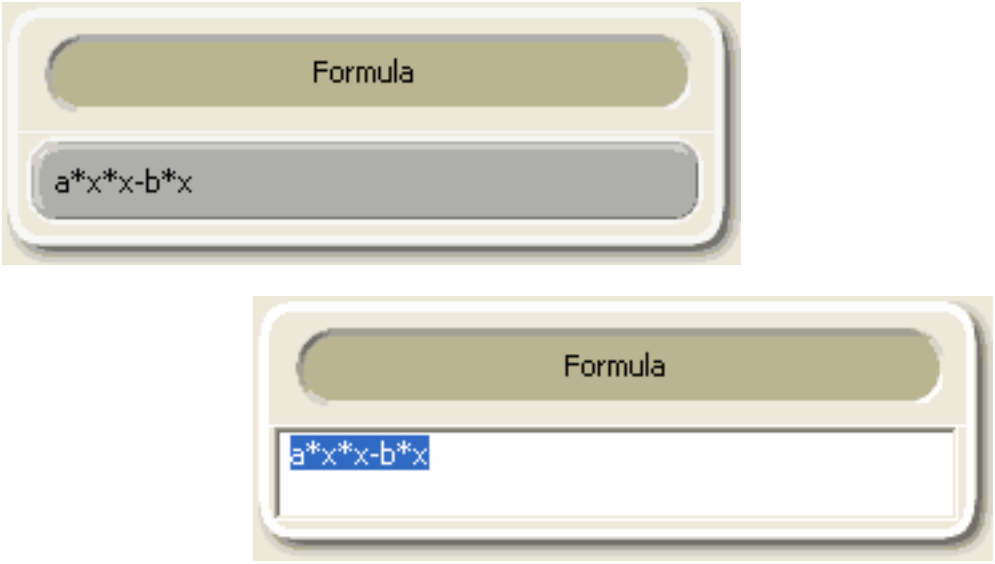


Fig. 1.60: Figure 60: Themed text editor, without focus and with focus

1.11.13 ThemedVerticalNotebookEditor()

Suitable for Lists of Instances

Default for (none)

Optional parameters *closed_theme, double_click, open_theme, page_name, multiple_open, scrollable, view*

The ThemedVerticalNotebookEditor() factory generates a "notebook" editor, containing tabs that can be vertically expanded or collapsed. It can be used for lists of instances, similarly to the ListEditor() factory, with the *use_notebook* parameter. You can specify themes to use for the open and closed states of the tabs.

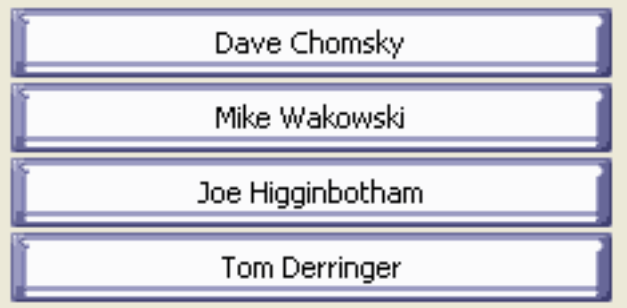


Fig. 1.61: Figure 61: Themed vertical notebook, with tabs for Person instances closed

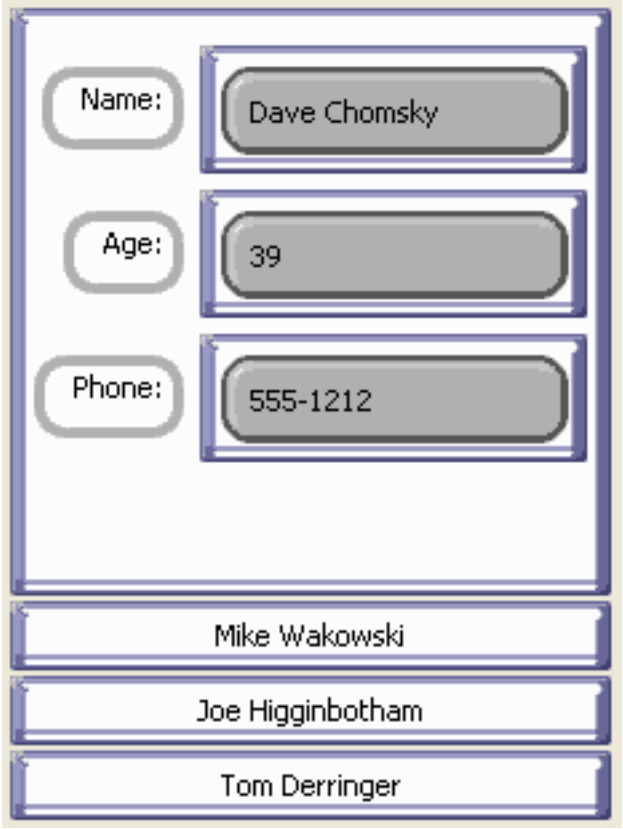


Fig. 1.62: Figure 62: Themed vertical notebook, with one tab open

1.12 Tips, Tricks and Gotchas

1.12.1 Getting and Setting Model View Elements

For some applications, it can be necessary to retrieve or manipulate the View objects associated with a given model object. The HasTraits class defines two methods for this purpose: trait_views() and trait_view().

trait_views()

The trait_views() method, when called without arguments, returns a list containing the names of all Views defined in the object's class. For example, if **sam** is an object of type SimpleEmployee3 (from *Example 6*), the method call `sam.trait_views()` returns the list `['all_view', 'traits_view']`.

Alternatively, a call to `trait_views(view_element_type)` returns a list of all named instances of class *view_element_type* defined in the object's class. The possible values of *view_element_type* are:

- *View*
- *Group*
- *Item*
- *ViewElement*
- ViewSubElement

Thus calling `trait_views(View)` is identical to calling `trait_views()`. Note that the call `sam.trait_views(Group)` returns an empty list, even though both of the Views defined in SimpleEmployee contain Groups. This is because only *named* elements are returned by the method.

Group and Item are both subclasses of ViewSubElement, while ViewSubElement and View are both subclasses of ViewElement. Thus, a call to `trait_views(ViewSubElement)` returns a list of named Items and Groups, while `trait_views(ViewElement)` returns a list of named Items, Groups and Views.

trait_view()

The trait_view() method is used for three distinct purposes:

- To retrieve the default View associated with an object
- To retrieve a particular named ViewElement (i.e., Item, Group or View)
- To define a new named ViewElement

For example:

- `obj.trait_view()` returns the default View associated with object *obj*. For example, `sam.trait_view()` returns the View object called `traits_view`. Note that unlike trait_views(), trait_view() returns the View itself, not its name.
- `obj.trait_view('my_view')` returns the view element named `my_view` (or None if `my_view` is not defined).
- `obj.trait_view('my_group', Group('a', 'b'))` defines a Group with the name `my_group`. Note that although this Group can be retrieved using `trait_view()`, its name does not appear in the list returned by `traits_view(Group)`. This is because `my_group` is associated with *obj* itself, rather than with its class.

1.13 Appendix I: Glossary of Terms

attribute An element of data that is associated with all instances of a given class, and is named at the class level. [19] In most cases, attributes are stored and assigned separately for each instance (for the exception, see *class attribute*). Synonyms include "data member" and "instance variable".

class attribute An element of data that is associated with a class, and is named at the class level. There is only one value for a class attribute, associated with the class itself. In contrast, for an instance *attribute*, there is a value associated with every instance of a class.

command button A button on a window that globally controls the window. Examples include *OK*, *Cancel*, *Apply*, *Revert*, and *Help*.

controller The element of the *MVC* ("model-view-controller") design pattern that manages the transfer of information between the data *model* and the *view* used to observe and edit it.

dialog box A secondary window whose purpose is for a user to specify additional information when entering a command.

editor A user interface component for editing the value of a trait attribute. Each type of trait has a default editor, but you can override this selection with one of a number of editor factories provided by the TraitsUI package. In some cases an editor can include multiple widgets, e.g., a slider and a text box for a Range trait attribute.

editor factory An instance of the Traits class EditorFactory. Editor factories generate the actual widgets used in a user interface. You can use an editor factory without knowing what the underlying GUI toolkit is.

factory An object used to produce other objects at run time without necessarily assigning them to named variables or attributes. A single factory is often parameterized to produce instances of different classes as needed.

Group An object that specifies an ordered set of Items and other Groups for display in a TraitsUI View. Various display options can be specified by means of attributes of this class, including a border, a group label, and the orientation of elements within the Group. An instance of the TraitsUI class Group.

Handler A TraitsUI object that implements GUI logic (data manipulation and dynamic window behavior) for one or more user interface windows. A Handler instance fills the role of *controller* in the MVC design pattern. An instance of the TraitsUI class *Handler*.

HasTraits A class defined in the Traits package to specify objects whose attributes are typed. That is, any attribute of a HasTraits subclass can be a *trait attribute*.

instance A concrete entity belonging to an abstract category such as a class. In object-oriented programming terminology, an entity with allocated memory storage whose structure and behavior are defined by the class to which it belongs. Often called an *object*.

Item A non-subdividable element of a Traits user interface specification (View), usually specifying the display options to be used for a single trait attribute. An instance of the TraitsUI class Item.

live A term used to describe a window that is linked directly to the underlying model data, so that changes to data in the interface are reflected immediately in the model. A window that is not live displays and manipulates a copy of the model data until the user confirms any changes.

livemodal A term used to describe a window that is both *live* and *modal*.

MVC A design pattern for interactive software applications. The initials stand for "Model-View-Controller", the three distinct entities prescribed for designing such applications. (See the glossary entries for *model*, *view*, and *controller*.)

modal A term used to describe a window that causes the remainder of the application to be suspended, so that the user can interact only with the window until it is closed.

[19] This is not always the case in Python, where attributes can be added to individual objects.

model A component of the *MVC* design pattern for interactive software applications. The model consists of the set of classes and objects that define the underlying data of the application, as well as any internal (i.e., non-GUI-related) methods or functions on that data.

nonmodal A term used to describe a window that is neither *live* nor *modal*.

object Synonym for *instance*.

panel A user interface region similar to a window except that it is embedded in a larger window rather than existing independently.

predefined trait type Any trait type that is built into the Traits package.

subpanel A variation on a *panel* that ignores (i.e., does not display) any command buttons.

trait A term used loosely to refer to either a *trait type* or a *trait attribute*.

trait attribute An *attribute* whose type is specified and checked by means of the Traits package.

trait type A type-checked data type, either built into or implemented by means of the Traits package.

Traits An open source package engineered by Enthought, Inc. to perform explicit typing in Python.

TraitsUI A high-level user interface toolkit designed to be used with the Traits package.

View A template object for constructing a GUI window or panel for editing a set of traits. The structure of a View is defined by one or more Group or Item objects; a number of attributes are defined for specifying display options including height and width, menu bar (if any), and the set of buttons (if any) that are displayed. A member of the TraitsUI class View.

view A component of the *MVC* design pattern for interactive software applications. The view component encompasses the visual aspect of the application, as opposed to the underlying data (the *model*) and the application's behavior (the *controller*).

ViewElement A View, Group or Item object. The ViewElement class is the parent of all three of these subclasses.

widget An interactive element in a graphical user interface, e.g., a scrollbar, button, pull-down menu or text box.

wizard An interface composed of a series of *dialog box* windows, usually used to guide a user through an interactive task such as software installation.

wx A shorthand term for the low-level GUI toolkit on which TraitsUI and PyFace are currently based (wxWidgets) and its Python wrapper (wxPython).

1.14 Appendix II: Editor Factories for Predefined Traits

Predefined traits that are not listed in this table use TextEditor() by default, and have no other appropriate editor factories.

Trait	Default Editor Factory	Other Possible Editor Factories
Any	TextEditor	EnumEditor, ImageEnumEditor, Valu
Array	ArrayEditor (for 2-D arrays)	
Bool	BooleanEditor	ThemedCheckboxEditor
Button	ButtonEditor	
CArray	ArrayEditor (for 2-D arrays)	
CBool	BooleanEditor	
CComplex	TextEditor	
CFloat, CInt, CLong	TextEditor	LEDEditor
Code	CodeEditor	

Table 1.1 – continued from previous page

Trait	Default Editor Factory	Other Possible Editor Factories
Color	ColorEditor	
Complex	TextEditor	
CStr, CUnicode	TextEditor (multi_line=True)	CodeEditor, HTMLEditor
Dict	TextEditor	ValueEditor
Directory	DirectoryEditor	
Enum	EnumEditor	ImageEnumEditor
Event	(none)	ButtonEditor, ToolbarButtonEditor
File	FileEditor	AnimatedGIFEditor
Float	TextEditor	LEDEditor
Font	FontEditor	
HTML	HTMLEditor	
Instance	InstanceEditor	TreeEditor, DropEditor, DNDEditor,
List	TableEditor for lists of HasTraits objects; ListEditor for all other lists.	CSVListEditor, CheckListEditor, Set
Long	TextEditor	LEDEditor
Password	TextEditor(password=True)	
PythonValue	ShellEditor	
Range	RangeEditor	ThemedSliderEditor
Regex	TextEditor	CodeEditor
RGBColor	RGBColorEditor	
Str	TextEditor(multi_line=True)	CodeEditor, HTMLEditor
String	TextEditor	CodeEditor, ThemedTextEditor
This	InstanceEditor	
ToolbarButton	ButtonEditor	
Tuple	TupleEditor	
UIDebugger	ButtonEditor (button calls the UIDebugEditor factory)	
Unicode	TextEditor(multi_line=True)	HTMLEditor
WeakRef	InstanceEditor	

CHAPTER 2

TraitsUI 4 Tutorials

2.1 Writing a graphical application for scientific programming using TraitsUI 4

A step by step guide for a non-programmer

Author Gael Varoquaux

Date 2015-08-18

License BSD

Building interactive Graphical User Interfaces (GUIs) is a hard problem, especially for somebody who has not had training in IT. TraitsUI is a python module that provides a great answer to this problem. I have found that I am incredibly productive when creating graphical application using traitsUI. However I had to learn a few new concepts and would like to lay them down together in order to make it easier for others to follow my footsteps.

This document is intended to help a non-programmer to use traits and traitsUI to write an interactive graphical application. The reader is assumed to have some basic python scripting knowledge (see ref [1] for a basic introduction). Knowledge of numpy/scipy [2] helps understanding the data processing aspects of the examples, but may not be paramount. Some examples rely on matplotlib [3] . This document is **not** a replacement for user manuals and references of the different packages (traitsUI [4], scipy, matplotlib). It provides a "cookbook" approach, and not a reference.

This tutorial provides step-by-step guide to building a medium-size application. The example chosen is an application used to do control of a camera, analysis of the retrieved data and display of the results. This tutorial focuses on building the general structure and flow-control of the application, and on the aspects specific to traitsUI programming. Interfacing with the hardware or processing the data is left aside. The tutorial progressively introduces the tools used, and in the end presents the skeleton of a real application that has been developed for real-time controlling of an experiment, monitoring through a camera, and processing the data. The tutorial goes into more and more intricate details that are necessary to build the final application. Each section is in itself independent of the following ones. The complete beginner trying to use this as an introduction should not expect to understand all the details in a first pass.

The author's experience while working on several projects in various physics labs is that code tends to be created in an 'organic' way, by different people with various levels of qualification in computer development, and that it rapidly decays to a disorganized and hard-to-maintain code base. This tutorial tries to prevent this by building an application shaped for modularity and readability.

[1] python tutorial: http://docs.python.org/tut/tut.html
[2] The scipy website: http://www.scipy.org
[3] The matplotlib website: http://matplotlib.sourceforge.net
[4] The traits and traitsUI user guide: http://code.enthought.com/traits

2.1.1 From objects to dialogs using traitsUI

Creating user interfaces directly through a toolkit is a time-consuming process. It is also a process that does not integrate well in the scientific-computing work-flow, as, during the elaboration of algorithms and data-flow, the objects that are represented in the GUI are likely to change often.

Visual computing, where the programmer creates first a graphical interface and then writes the callbacks of the graphical objects, gives rise to a slow development cycle, as the work-flow is centered on the GUI, and not on the code.

TraitsUI provides a beautiful answer to this problem by building graphical representations of an object. Traits and TraitsUI have their own manuals (http://code.enthought.com/traits/) and the reader is encouraged to refer to these for more information.

We will use TraitsUI for *all* our GUIs. This forces us to store all the data and parameters in objects, which is good programming style. The GUI thus reflects the structure of the code, which makes it easier to understand and extend.

In this section we will focus on creating dialogs that allow the user to input parameters graphically in the program.

Object-oriented programming

Software engineering is a difficult field. As programs, grow they become harder and harder to grasp for the developer. This problem is not new and has sometimes been know as the "tar pit". Many attempts have been made to mitigate the difficulties. Most often they consist in finding useful abstractions that allow the developer to manipulate larger ideas, rather than their software implementation.

Code re-use is paramount for good software development. It reduces the number of code-lines required to read and understand and allows to identify large operations in the code. Functions and procedures have been invented to avoid copy-and-pasting code, and hide the low-level details of an operation.

Object-oriented programming allows yet more modularity and abstraction.

Objects, attributes and methods

Suppose you want your program to manipulate geometric objects. You can teach the computer that a point is a set of 3 numbers, you can teach it how to rotate that point along a given axis. Now you want to use spheres too. With a bit more work your program has functions to create points, spheres, etc. It knows how to rotate them, to mirror them, to scale them. So in pure procedural programming you will have procedures to rotate, scale, mirror, each one of your objects. If you want to rotate an object you will first have to find its type, then apply the right procedure to rotate it.

Object-oriented programming introduces a new abstraction: the *object*. It consists of both data (our 3 numbers, in the case of a point), and procedures that use and modify this data (e.g., rotations). The data entries are called "*attributes*" of the object and the procedures "*methods*". Thus with object oriented programming an object "knows" how to be rotated.

A point object could be implemented in python with:

code snippet #0

```
from numpy import cos, sin

class Point(object):
    """ 3D Point objects """
    x = 0.
    y = 0.
    z = 0.

    def rotate_z(self, theta):
        """ rotate the point around the Z axis """
```

```
            xtemp =  cos(theta) * self.x + sin(theta) * self.y
            ytemp = -sin(theta) * self.x + cos(theta) * self.y
            self.x = xtemp
            self.y = ytemp
```

This code creates a *Point* class. Points objects can be created as *instances* of the Point class:

```
>>> from numpy import pi
>>> p = Point()
>>> p.x = 1
>>> p.rotate_z(pi)
>>> p.x
-1.0
>>> p.y
1.2246467991473532e-16
```

When manipulating objects, the developer does not need to know the internal details of their procedures. As long as the object has a *rotate* method, the developer knows how to rotate it.

> **Note**: Beginners often use objects as structures: entities with several data fields useful to pass data around in a program. Objects are much more then that: they have methods. They are 'active' data structures that know how to modify themselves. Part of the point of object-oriented programming is that the object is responsible for modifying itself through its methods. The object therefore takes care of its internal logic and the consistency between its attributes.
>
> In python, dictionaries make great structures and are more suited for such a use than objects.

Classes and inheritance

Suppose you have already created a *Point* class that tells your program what a point is, but that you also want some points to have a color. Instead of copy-and-pasting the *Point* class and adding a color attribute, you can define a new class *ColoredPoint* that inherits all of the *Point* class's methods and attributes:

```
class ColoredPoint(Point):
    """ Colored 3D point """
    color = "white"
```

You do not have to implement rotation for the *ColoredPoint* class as it has been inherited from the *Point* class. This is one of the huge gains of object-oriented programming: objects are organized in classes and sub-classes, and method to manipulate objects are derived from the objects parent-ship: a *ColoredPoint* is only a special case of *Point*. This proves very handy on large projects.

> **Note**: To stress the differences between classes and their instances (objects), classes are usually named with capital letters, and objects only with lower case letters.

An object and its representation

Objects are code entities that can be easily pictured by the developer. The *TraitsUI* python module allows the user to edit objects attributes with dialogs that form a graphical representation of the object.

In our example application, each process or experimental device is represented in the code as an object. These objects all inherit from the *HasTraits*, class which supports creating graphical representations of attributes. To be able to build the dialog, the *HasTraits* class enforces that the types of all the attributes are specified in the class definition.

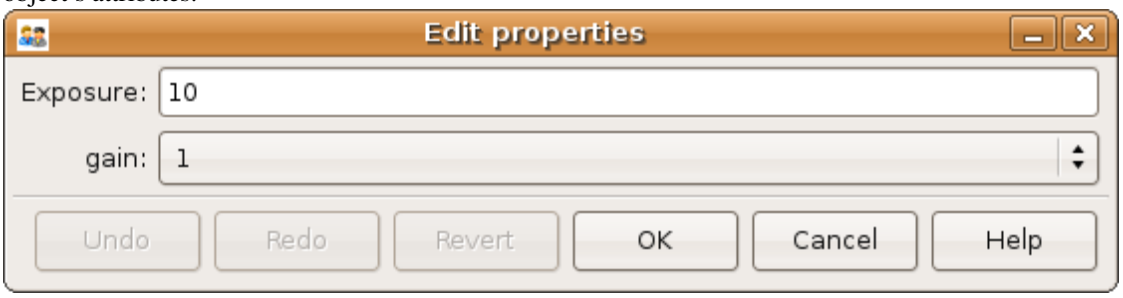
The *HasTraits* objects have a *configure_traits()* method that brings up a dialog to edit the objects' attributes specified in its class definition.

Here we define a camera object (which, in our real world example, is a camera interfaced to python through the ctypes [5] module), and show how to open a dialog to edit its properties :

code snippet #1

```
from traits.api import *
from traitsui.api import *

class Camera(HasTraits):
    """ Camera object """

    gain = Enum(1, 2, 3,
        desc="the gain index of the camera",
        label="gain", )

    exposure = CInt(10,
        desc="the exposure time, in ms",
        label="Exposure", )

    def capture(self):
        """ Captures an image on the camera and returns it """
        print "capturing an image at %i ms exposure, gain: %i" % (
                self.exposure, self.gain )

if __name__ == "__main__":
    camera = Camera()
    camera.configure_traits()
    camera.capture()
```

The *camera.configure_traits()* call in the above example opens a dialog that allows the user to modify the camera object's attributes:



This dialog forms a graphical representation of our camera object. We will see that it can be embedded in GUI panels to build more complex GUIs that allow us to control many objects.

We will build our application around objects and their graphical representation, as this mapping of the code to the GUI helps the developer to understand the code.

Displaying several objects in the same panel

We now know how to build a dialog from objects. If we want to build a complex application we are likely to have several objects, for instance one corresponding to the camera we want to control, and one describing the experiment that the camera monitors. We do not want to have to open a new dialog per object: this would force us to describe the GUI in terms of graphical objects, and not structural objects. We want the GUI to be a natural representation of our objects, and we want the Traits module to take care of that.

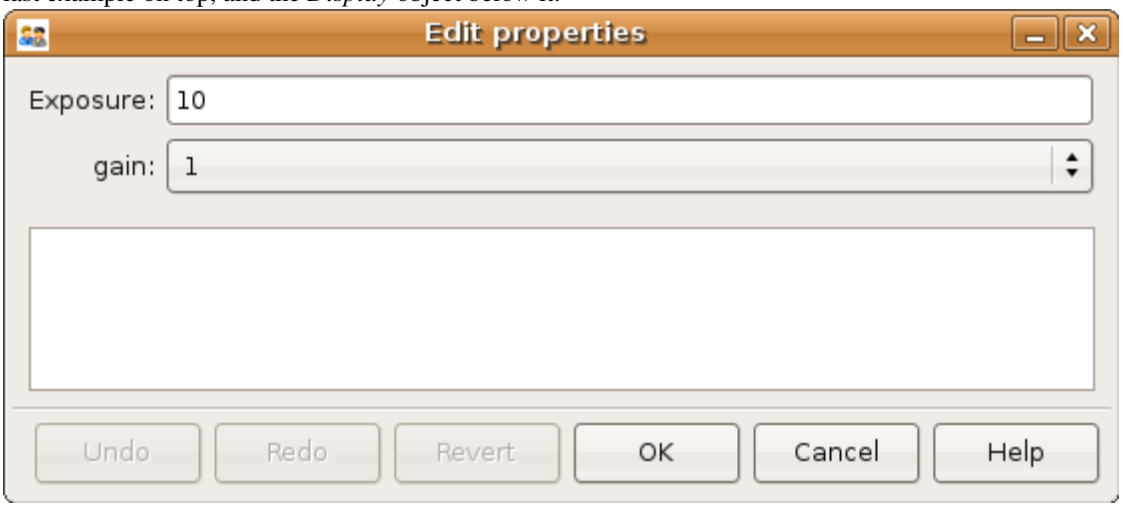
[5] ctypes: http://starship.python.net/crew/theller/ctypes/

The solution is to create a container object, that has as attributes the objects we want to represent. Playing with the *View* attribute of the object, we can control how the representation generated by Traits looks like (see the TraitsUI manual):

code snippet #2

```
from traits.api import *
from traitsui.api import *

class Camera(HasTraits):
    gain = Enum(1, 2, 3, )
    exposure = CInt(10, label="Exposure", )

class TextDisplay(HasTraits):
    string = String()

    view= View( Item('string', show_label=False, springy=True, style='custom' ))

class Container(HasTraits):
    camera = Instance(Camera)
    display = Instance(TextDisplay)

    view = View(
                Item('camera', style='custom', show_label=False, ),
                Item('display', style='custom', show_label=False, ),
            )

container = Container(camera=Camera(), display=TextDisplay())
container.configure_traits()
```

The call to *configure_traits()* creates the following dialog, with the representation of the *Camera* object created is the last example on top, and the *Display* object below it:



The *View* attribute of the *container* object has been tweaked to get the representation we are interested in: traitsUI is told to display the *camera* item with a *'custom'* style, which instructs it to display the representation of the object inside the current panel. The *'show_label'* argument is set to *False* as we do not want the name of the displayed object ('camera', for instance) to appear in the dialog. See the traitsUI manual for more details on this powerful feature.

The *camera* and *display* objects are created during the call to the creator of the *container* object, and passed as its attributes immediately: *"container = Container(camera=Camera(), display=TextDisplay())"*

Writing a "graphical script"

If you want to create an application that has a very linear flow, popping up dialogs when user input is required, like a "setup wizard" often used to install programs, you already have all the tools to do it. You can use object oriented programming to write your program, and call the objects *configure_traits* method each time you need user input. This might be an easy way to modify an existing script to make it more user friendly.

The following section will focus on making interactive programs, where the user uses the graphical interface to interact with it in a continuous way.

2.1.2 From graphical to interactive

In an interactive application, the program responds to user interaction. This requires a slight paradigm shift in our programming methods.

Object-oriented GUIs and event loops

In a GUI application, the order in which the different parts of the program are executed is imposed by the user, unlike in a numerical algorithm, for instance, where the developer chooses the order of execution of his program. An event loop allows the programmer to develop an application in which each user action triggers an event, by stacking the user created events on a queue, and processing them in the order in which the appeared.

A complex GUI is made of a large numbers of graphical elements, called widgets (e.g., text boxes, check boxes, buttons, menus). Each of these widgets has specific behaviors associated with user interaction (modifying the content of a text box, clicking on a button, opening a menu). It is natural to use objects to represent the widgets, with their behavior being set in the object's methods.

Dialogs populated with widgets are automatically created by *traitsUI* in the *configure_traits()* call. *traitsUI* allow the developer to not worry about widgets, but to deal only with objects and their attributes. This is a fabulous gain as the widgets no longer appear in the code, but only the attributes they are associated to.

A *HasTraits* object has an *edit_traits()* method that creates a graphical panel to edit its attributes. This method creates and returns the panel, but does not start its event loop. The panel is not yet "alive", unlike with the *configure_traits()* method. Traits uses the wxWidget toolkit by default to create its widget. They can be turned live and displayed by starting a wx application, and its main loop (ie event loop in wx speech).

code snippet #3

```
from traits.api import *
import wx

class Counter(HasTraits):
    value =  Int()

Counter().edit_traits()
wx.PySimpleApp().MainLoop()
```

The *Counter().edit_traits()* line creates a counter object and its representation, a dialog with one integer represented. However it does not display it until a wx application is created, and its main loop is started.

Usually it is not necessary to create the wx application yourself, and to start its main loop, traits will do all this for you when the *.configure_traits()* method is called.

Reactive programming

When the event loop is started, the program flow is no longer simply controlled by the code: the control is passed on to the event loop, and it processes events, until the user closes the GUI, and the event loop returns to the code.

Interactions with objects generate events, and these events can be associated to callbacks, ie functions or methods processing the event. In a GUI, callbacks created by user-generated events are placed on an "event stack". The event loop processes each call on the event queue one after the other, thus emptying the event queue. The flow of the program is still sequential (two code blocks never run at the same time in an event loop), but the execution order is chosen by the user, and not by the developer.

Defining callbacks for the modification of an attribute *foo* of a *HasTraits* object can be done be creating a method called *_foo_changed()*. Here is an example of a dialog with two textboxes, *input* and *output*. Each time *input* is modified, is content is duplicated to output.

code snippet #4

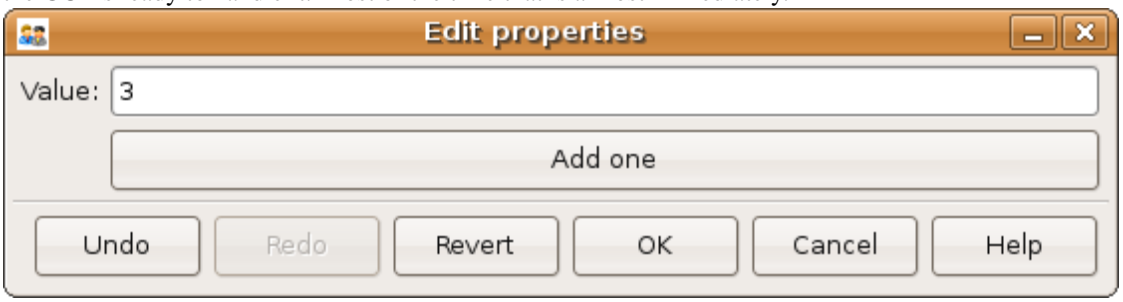
```
from traits.api import *

class EchoBox(HasTraits):
    input =  Str()
    output = Str()

    def _input_changed(self):
        self.output = self.input

EchoBox().configure_traits()
```

Events that do not correspond to a modification of an attribute can be generated with a *Button* traits. The callback is then called *_foo_fired()*. Here is an example of an interactive *traitsUI* application using a button:

code snippet #5

```
from traits.api import *
from traitsui.api import View, Item, ButtonEditor

class Counter(HasTraits):
    value =  Int()
    add_one = Button()

    def _add_one_fired(self):
        self.value +=1

    view = View('value', Item('add_one', show_label=False ))

Counter().configure_traits()
```

Clicking on the button adds the *_add_one_fired()* method to the event queue, and this method gets executed as soon as the GUI is ready to handle it. Most of the time that is almost immediately.



This programming pattern is called *reactive programming*: the objects react to the changes made to their attributes. In complex programs where the order of execution is hard to figure out, and bound to change, like some interactive data processing application, this pattern is extremely efficient.

Using *Button* traits and a clever set of objects interacting with each others, complex interactive applications can be built. These applications are governed by the events generated by the user, in contrast to script-like applications (batch programming). Executing a long operation in the event loop blocks the reactions of the user-interface, as other events callbacks are not processed as long as the long operation is not finished. In the next section we will see how we can execute several operations in the same time.

2.1.3 Breaking the flow in multiple threads

What are threads ?

A standard python program executes in a sequential way. Consider the following code snippet :

```
do_a()
do_b()
do_c()
```

do_b() is not called until *do_a()* is finished. Even in event loops everything is sequential. In some situation this can be very limiting. Suppose we want to capture an image from a camera and that it is a very lengthy operation. Suppose also that no other operation in our program requires the capture to be complete. We would like to have a different "timeline" in which the camera capture instructions can happen in a sequential way, while the rest of the program continues in parallel.

Threads are the solution to this problem: a thread is a portion of a program that can run concurrently with other portions of the program.

Programming with threads is difficult as instructions are no longer executed in the order they are specified and the output of a program can vary from a run to another, depending on subtle timing issues. These problems are known as "race conditions" and to minimize them you should avoid accessing the same objects in different threads. Indeed if two different threads are modifying the same object at the same time, unexpected things can happen.

Threads in python

In python a thread can be implemented with a *Thread* object, from the threading [6] module. To create your own execution thread, subclass the *Thread* object and put the code that you want to run in a separate thread in its *run* method. You can start your thread using its *start* method:

code snippet #6

```
from threading import Thread
from time import sleep

class MyThread(Thread):
    def run(self):
        sleep(2)
        print "MyThread done"

my_thread = MyThread()
```

[6] threading: http://docs.python.org/lib/module-threading.html

```
    my_thread.start()
    print "Main thread done"
```

The above code yields the following output:

```
Main thread done
MyThread done
```

Getting threads and the GUI event loop to play nice

Suppose you have a long-running job in a TraitsUI application. If you implement this job as an event placed on the event loop stack, it is going to freeze the event loop while running, and thus freeze the UI, as events will accumulate on the stack, but will not be processed as long as the long-running job is not done (remember, the event loop is sequential). To keep the UI responsive, a thread is the natural answer.

Most likely you will want to display the results of your long-running job on the GUI. However, as usual with threads, one has to be careful not to trigger race-conditions. Naively manipulating the GUI objects in your thread will lead to race conditions, and unpredictable crash: suppose the GUI was repainting itself (due to a window move, for instance) when you modify it.

In a wxPython application, if you start a thread, GUI event will still be processed by the GUI event loop. To avoid collisions between your thread and the event loop, the proper way of modifying a GUI object is to insert the modifications in the event loop, using the *GUI.invoke_later()* call. That way the GUI will apply your instructions when it has time.

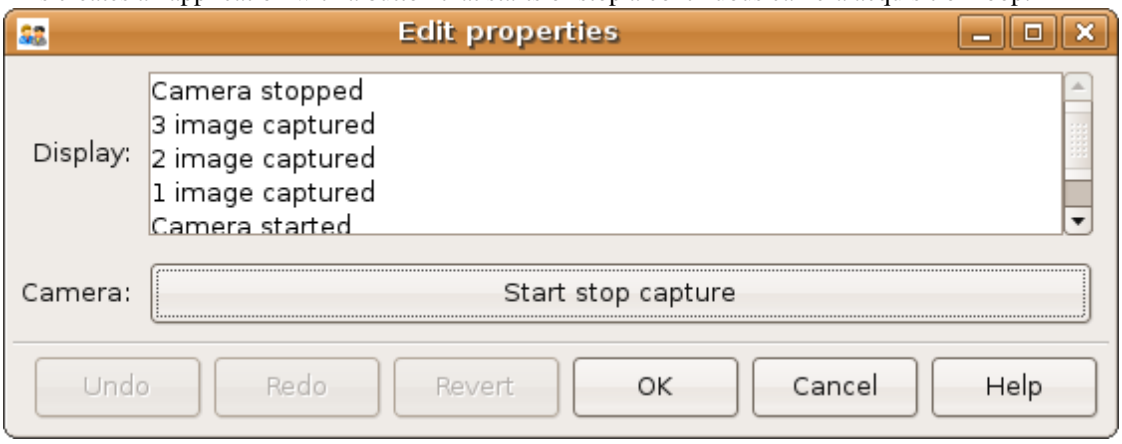
Recent versions of the TraitsUI module (post October 2006) propagate the changes you make to a *HasTraits* object to its representation in a thread-safe way. However it is important to have in mind that modifying an object with a graphical representation is likely to trigger race-conditions as it might be modified by the graphical toolkit while you are accessing it. Here is an example of code inserting the modification to traits objects by hand in the event loop:

code snippet #7

```
    from threading import Thread
    from time import sleep
    from traits.api import *
    from traitsui.api import View, Item, ButtonEditor

    class TextDisplay(HasTraits):
        string =  String()

        view= View( Item('string',show_label=False, springy=True, style='custom' ))


    class CaptureThread(Thread):
        def run(self):
            self.display.string = 'Camera started\n' + self.display.string
            n_img = 0
            while not self.wants_abort:
                sleep(.5)
                n_img += 1
                self.display.string = '%d image captured\n' % n_img \
                                                    + self.display.string
            self.display.string = 'Camera stopped\n' + self.display.string

    class Camera(HasTraits):
        start_stop_capture = Button()
        display = Instance(TextDisplay)
```

```
        capture_thread = Instance(CaptureThread)

        view = View( Item('start_stop_capture', show_label=False ))

        def _start_stop_capture_fired(self):
            if self.capture_thread and self.capture_thread.isAlive():
                self.capture_thread.wants_abort = True
            else:
                self.capture_thread = CaptureThread()
                self.capture_thread.wants_abort = False
                self.capture_thread.display = self.display
                self.capture_thread.start()

    class MainWindow(HasTraits):
        display = Instance(TextDisplay, ())

        camera = Instance(Camera)

        def _camera_default(self):
            return Camera(display=self.display)

        view = View('display', 'camera', style="custom", resizable=True)


    if __name__ == '__main__':
        MainWindow().configure_traits()
```

This creates an application with a button that starts or stop a continuous camera acquisition loop.



When the "Start stop capture" button is pressed the *_start_stop_capture_fired* method is called. It checks to see if a *CaptureThread* is running or not. If none is running, it starts a new one. If one is running, it sets its *wants_abort* attribute to true.

The thread checks every half a second to see if its attribute *wants_abort* has been set to true. If this is the case, it aborts. This is a simple way of ending the thread through a GUI event.

Using different threads lets the operations avoid blocking the user interface, while also staying responsive to other events. In the real-world application that serves as the basis of this tutorial, there are 2 threads and a GUI event loop.

The first thread is an acquisition loop, during which the program loops, waiting for a image to be captured on the camera (the camera is controlled by external signals). Once the image is captured and transfered to the computer, the

acquisition thread saves it to the disk and spawns a thread to process the data, then returns to waiting for new data while the processing thread processes the data. Once the processing thread is done, it displays its results (by inserting the display events in the GUI event loop) and dies. The acquisition thread refuses to spawn a new processing thread if there still is one running. This makes sure that data is never lost, no matter how long the processing might be.

There are thus up to 3 set of instructions running concurrently: the GUI event loop, responding to user-generated events, the acquisition loop, responding to hardware-generated events, and the processing jobs, doing the numerical intensive work.

In the next section we are going to see how to add a home-made element to traits, in order to add new possibilities to our application.

2.1.4 Extending TraitsUI: Adding a matplotlib figure to our application

This section gives a few guidelines on how to build your own traits editor. A traits editor is the view associated with a trait that allows the user to graphically edit its value. We can twist a bit the notion and simply use it to graphically represent the attribute. This section involves a bit of *wxPython* code that may be hard to understand if you do not know *wxPython*, but it will bring a lot of power and flexibility to how you use traits. The reason it appears in this tutorial is that I wanted to insert a matplotlib in my *traitsUI* application. It is not necessary to fully understand the code of this section to be able to read on.

I should stress that there already exists a plotting module that provides traits editors for plotting, and that is very well integrated with traits: chaco [7].

Making a *traits* editor from a MatPlotLib plot

To use traits, the developer does not need to know its internals. However traits does not provide an editor for every need. If we want to insert a powerful tool for plotting we have to get our hands a bit dirty and create our own traits editor.

This involves some *wxPython* coding, as we need to translate a *wxPython* object to a traits editor by providing the corresponding API (i.e. the standard way of building a *traits* editor), so that the *traits* framework will know how to create the editor.

Traits editor are created by an editor factory that instantiates an editor class and passes it the object that the editor represents in its *value* attribute. It calls the editor *init()* method to create the *wx* widget. Here we create a wx figure canvas from a matplotlib figure using the matplotlib wx backend. Instead of displaying this widget, we set its control as the *control* attribute of the editor. TraitsUI takes care of displaying and positioning the editor.

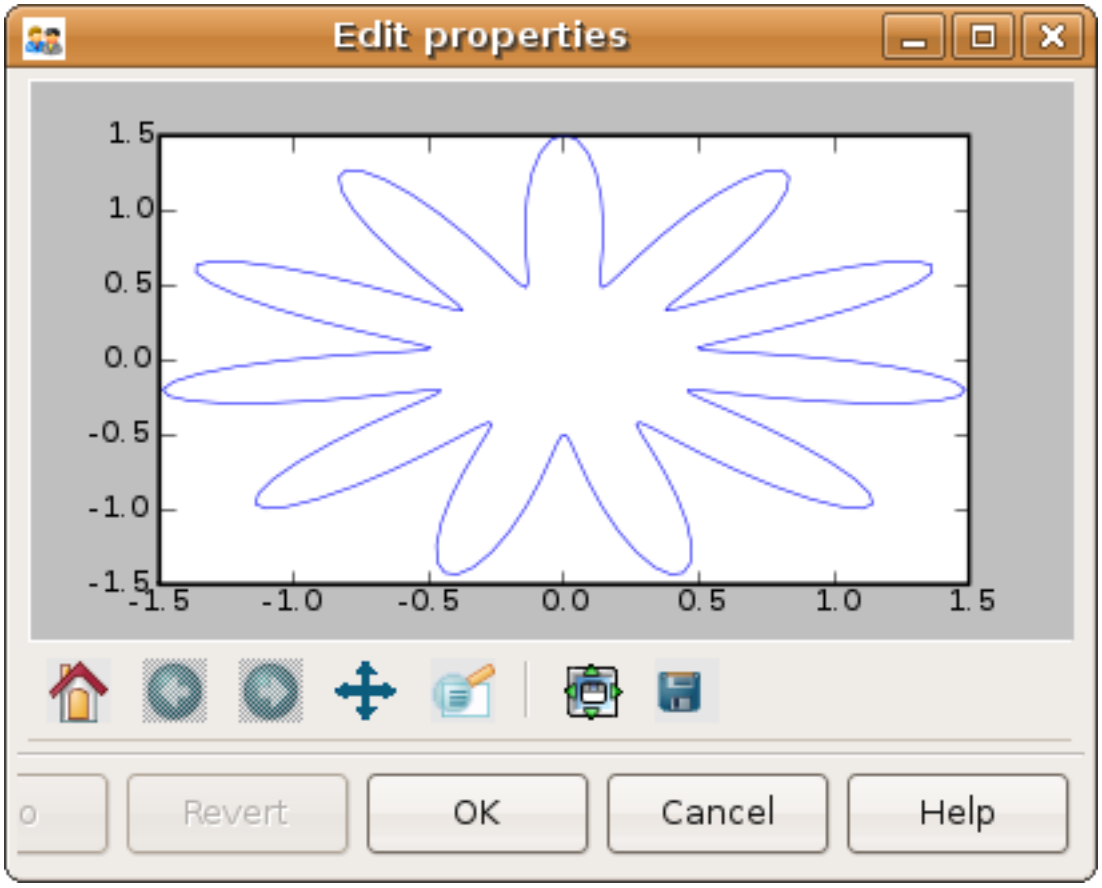
code snippet #8

```
import wx

import matplotlib
# We want matplotlib to use a wxPython backend
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.figure import Figure
from matplotlib.backends.backend_wx import NavigationToolbar2Wx

from traits.api import Any, Instance
from traitsui.wx.editor import Editor
from traitsui.wx.basic_editor_factory import BasicEditorFactory

class _MPLFigureEditor(Editor):
```

[7] chaco: http://code.enthought.com/chaco/

```
        scrollable  = True

        def init(self, parent):
            self.control = self._create_canvas(parent)
            self.set_tooltip()

        def update_editor(self):
            pass

        def _create_canvas(self, parent):
            """ Create the MPL canvas. """
            # The panel lets us add additional controls.
            panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
            sizer = wx.BoxSizer(wx.VERTICAL)
            panel.SetSizer(sizer)
            # matplotlib commands to create a canvas
            mpl_control = FigureCanvas(panel, -1, self.value)
            sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
            toolbar = NavigationToolbar2Wx(mpl_control)
            sizer.Add(toolbar, 0, wx.EXPAND)
            self.value.canvas.SetMinSize((10,10))
            return panel

    class MPLFigureEditor(BasicEditorFactory):

        klass = _MPLFigureEditor


    if __name__ == "__main__":
        # Create a window to demo the editor
        from traits.api import HasTraits
        from traitsui.api import View, Item
        from numpy import sin, cos, linspace, pi

        class Test(HasTraits):

            figure = Instance(Figure, ())

            view = View(Item('figure', editor=MPLFigureEditor(),
                                       show_label=False),
                            width=400,
                            height=300,
                            resizable=True)

            def __init__(self):
                super(Test, self).__init__()
                axes = self.figure.add_subplot(111)
                t = linspace(0, 2*pi, 200)
                axes.plot(sin(t)*(1+0.5*cos(11*t)), cos(t)*(1+0.5*cos(11*t)))

        Test().configure_traits()
```

This code first creates a traitsUI editor for a matplotlib figure, and then a small dialog to illustrate how it works:



The matplotlib figure traits editor created in the above example can be imported in a traitsUI application and combined with the power of traits. This editor allows to insert a matplotlib figure in a traitsUI dialog. It can be modified using reactive programming, as demonstrated in section 3 of this tutorial. However, once the dialog is up and running, you have to call *self.figure.canvas.draw()* to update the canvas if you made modifications to the figure. The matplotlib user guide [3] details how this object can be used for plotting.

2.1.5 Putting it all together: a sample application

The real world problem that motivated the writing of this tutorial is an application that retrieves data from a camera, processes it and displays results and controls to the user. We now have all the tools to build such an application. This section gives the code of a skeleton of this application. This application actually controls a camera on a physics experiment (Bose-Einstein condensation), at the university of Toronto.

The reason I am providing this code is to give an example to study of how a full-blown application can be built. This code can be found in the tutorial's zip file (it is the file *application.py*).

- The camera will be built as an object. Its real attributes (exposure time, gain...) will be represented as the object's attributes, and exposed through traitsUI.
- The continuous acquisition/processing/user-interaction will be handled by appropriate threads, as discussed in section 2.3.
- The plotting of the results will be done through the MPLWidget object.

The imports

The MPLFigureEditor is imported from the last example.

```
from threading import Thread
from time import sleep
from traits.api import *
from traitsui.api import View, Item, Group, HSplit, Handler
from traitsui.menu import NoButtons
from mpl_figure_editor import MPLFigureEditor
from matplotlib.figure import Figure
from scipy import *
import wx
```

User interface objects

These objects store information for the program to interact with the user via traitsUI.

```
class Experiment(HasTraits):
    """ Object that contains the parameters that control the experiment,
    modified by the user.
    """
    width = Float(30, label="Width", desc="width of the cloud")
    x = Float(50, label="X", desc="X position of the center")
    y = Float(50, label="Y", desc="Y position of the center")

class Results(HasTraits):
    """ Object used to display the results.
    """
    width = Float(30, label="Width", desc="width of the cloud")
    x = Float(50, label="X", desc="X position of the center")
    y = Float(50, label="Y", desc="Y position of the center")

    view = View( Item('width', style='readonly'),
                 Item('x', style='readonly'),
                 Item('y', style='readonly'),
               )
```

The camera object also is a real object, and not only a data structure: it has a method to acquire an image (or in our case simulate acquiring), using its attributes as parameters for the acquisition.

```
class Camera(HasTraits):
    """ Camera objects. Implements both the camera parameters controls, and
    the picture acquisition.
    """
    exposure = Float(1, label="Exposure", desc="exposure, in ms")
    gain = Enum(1, 2, 3, label="Gain", desc="gain")

    def acquire(self, experiment):
        X, Y = indices((100, 100))
        Z = exp(-((X-experiment.x)**2+(Y-experiment.y)**2)/experiment.width**2)
        Z += 1-2*rand(100,100)
        Z *= self.exposure
        Z[Z>2] = 2
        Z = Z**self.gain
        return(Z)
```

Threads and flow control

There are three threads in this application:

- The GUI event loop, the only thread running at the start of the program.
- The acquisition thread, started through the GUI. This thread is an infinite loop that waits for the camera to be triggered, retrieves the images, displays them, and spawns the processing thread for each image received.
- The processing thread, started by the acquisition thread. This thread is responsible for the numerical intensive work of the application. It processes the data and displays the results. It dies when it is done. One processing thread runs per shot acquired on the camera, but to avoid accumulation of threads in the case that the processing takes longer than the time lapse between two images, the acquisition thread checks that the processing thread is done before spawning a new one.

```
        def process(image, results_obj):
            """ Function called to do the processing """
            X, Y = indices(image.shape)
            x = sum(X*image)/sum(image)
            y = sum(Y*image)/sum(image)
            width = sqrt(abs(sum(((X-x)**2+(Y-y)**2)*image)/sum(image)))
            results_obj.x = x
            results_obj.y = y
            results_obj.width = width

        class AcquisitionThread(Thread):
            """ Acquisition loop. This is the worker thread that retrieves images
            from the camera, displays them, and spawns the processing job.
            """
            wants_abort = False

            def process(self, image):
                """ Spawns the processing job. """
                try:
                    if self.processing_job.isAlive():
                        self.display("Processing too slow")
                        return
                except AttributeError:
                    pass
                self.processing_job = Thread(target=process, args=(image,
                                             self.results))
                self.processing_job.start()

            def run(self):
                """ Runs the acquisition loop. """
                self.display('Camera started')
                n_img = 0
                while not self.wants_abort:
                    n_img += 1
                    img =self.acquire(self.experiment)
                    self.display('%d image captured' % n_img)
                    self.image_show(img)
                    self.process(img)
                    sleep(1)
                self.display('Camera stopped')
```

The GUI elements

The GUI of this application is separated in two (and thus created by a sub-class of SplitApplicationWindow).

On the left a plotting area, made of an MPL figure and its editor, displays the images acquired by the camera.

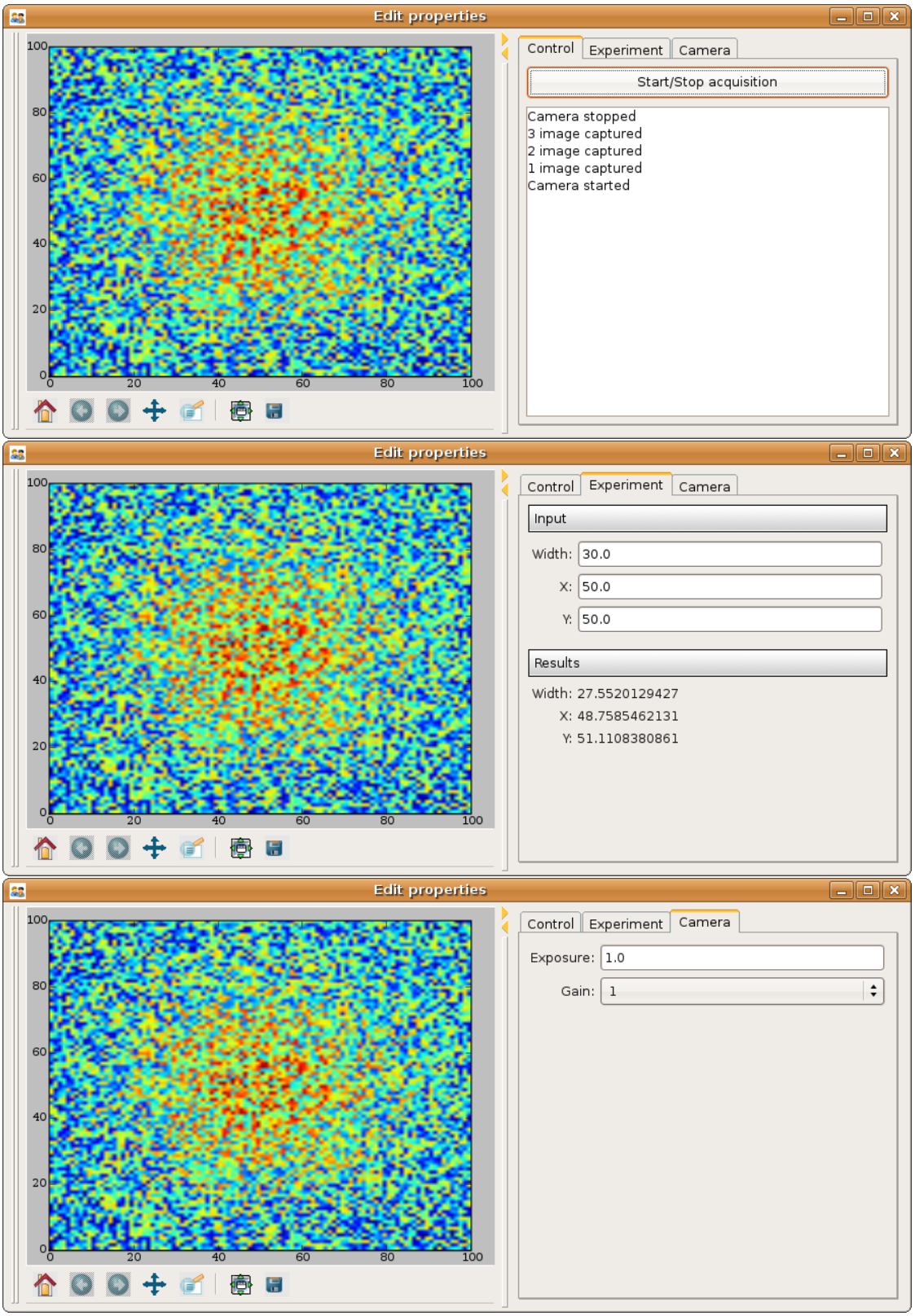
On the right a panel hosts the TraitsUI representation of a ControlPanel object. This object is mainly a container for our other objects, but it also has an Button for starting or stopping the acquisition, and a string (represented by a textbox) to display information on the acquisition process. The view attribute is tweaked to produce a pleasant and usable dialog. Tabs are used to help the display to be light and clear.

```
    class ControlPanel(HasTraits):
        """ This object is the core of the traitsUI interface. Its view is
        the right panel of the application, and it hosts the method for
        interaction between the objects and the GUI.
        """
        experiment = Instance(Experiment, ())
        camera = Instance(Camera, ())
        figure = Instance(Figure)
        results = Instance(Results, ())
        start_stop_acquisition = Button("Start/Stop acquisition")
        results_string = String()
        acquisition_thread = Instance(AcquisitionThread)
        view = View(Group(
                        Group(
                            Item('start_stop_acquisition', show_label=False ),
                            Item('results_string',show_label=False,
                                springy=True, style='custom' ),
                            label="Control", dock='tab',),
                        Group(
                            Group(
                                Item('experiment', style='custom', show_label=False),
                                label="Input",),
                            Group(
                                Item('results', style='custom', show_label=False),
                                label="Results",),
                            label='Experiment', dock="tab"),
                        Item('camera', style='custom', show_label=False, dock="tab"),
                        layout='tabbed'),
                        )

        def _start_stop_acquisition_fired(self):
            """ Callback of the "start stop acquisition" button. This starts
            the acquisition thread, or kills it.
            """
            if self.acquisition_thread and self.acquisition_thread.isAlive():
                self.acquisition_thread.wants_abort = True
            else:
                self.acquisition_thread = AcquisitionThread()
                self.acquisition_thread.display = self.add_line
                self.acquisition_thread.acquire = self.camera.acquire
                self.acquisition_thread.experiment = self.experiment
                self.acquisition_thread.image_show = self.image_show
                self.acquisition_thread.results = self.results
                self.acquisition_thread.start()

        def add_line(self, string):
            """ Adds a line to the textbox display.
            """
```

```
            self.results_string = (string + "\n" + self.results_string)[0:1000]

        def image_show(self, image):
            """ Plots an image on the canvas in a thread safe way.
            """
            self.figure.axes[0].images=[]
            self.figure.axes[0].imshow(image, aspect='auto')
            wx.CallAfter(self.figure.canvas.draw)

    class MainWindowHandler(Handler):
        def close(self, info, is_OK):
            if ( info.object.panel.acquisition_thread
                and info.object.panel.acquisition_thread.isAlive() ):
                info.object.panel.acquisition_thread.wants_abort = True
                while info.object.panel.acquisition_thread.isAlive():
                    sleep(0.1)
                wx.Yield()
            return True

    class MainWindow(HasTraits):
        """ The main window, here go the instructions to create and destroy the application. """
        figure = Instance(Figure)

        panel = Instance(ControlPanel)

        def _figure_default(self):
            figure = Figure()
            figure.add_axes([0.05, 0.04, 0.9, 0.92])
            return figure

        def _panel_default(self):
            return ControlPanel(figure=self.figure)

        view = View(HSplit(Item('figure', editor=MPLFigureEditor(),
                                dock='vertical'),
                           Item('panel', style="custom"),
                           show_labels=False,
                          ),
                    resizable=True,
                    height=0.75, width=0.75,
                    handler=MainWindowHandler(),
                    buttons=NoButtons)

    if __name__ == '__main__':
        MainWindow().configure_traits()
```

When the acquisition loop is created and running, the mock camera object produces noisy gaussian images, and the processing code estimates the parameters of the gaussian.

Here are screenshots of the three different tabs of the application:


Edit properties
Control
Experiment
Camera
Start/Stop acquisition
Camera stopped
3 image captured
2 image captured
1 image captured
Camera started
Edit properties
Control
Experiment
Camera
Input
Width: 30.0
X: 50.0
Y: 50.0
Results
Width: 27.5520129427
X: 48.7585462131
Y: 51.1108380861
Edit properties
Control
Experiment
Camera
Exposure: 1.0
Gain: 1

Conclusion

I have summarized here all what most scientists need to learn in order to be able to start building applications with traitsUI. Using the traitsUI module to its full power requires you to move away from the procedural type of programming most scientists are used to, and think more in terms of objects and flow of information and control between them. I have found that this paradigm shift, although a bit hard, has been incredibly rewarding in terms of my own productivity and my ability to write compact and readable code.

Good luck!

Acknowledgments

I would like to thank the people on the enthought-dev mailing-list, especially Prabhu Ramachandran and David Morrill, for all the help they gave me, and Janet Swisher for reviewing this document. Big thanks go to enthought for developing the traits and traitsUI modules, and making them open-source. Finally the python, the numpy, and the matplotlib community deserve many thanks for both writing such great software, and being so helpful on the mailing lists.

References

CHAPTER 3

TraitsUI 4 Demos

This section contains links to a number of TraitsUI demos.

Warning: Some of the examples in this section may be out of date. We are in the process of updating them to the latest version of TraitsUI.

3.1 Standard Editors

- BooleanEditor
- ButtonEditor
- CSVListEditor
- CheckListEditor
- CheckListEditor (simple)
- CodeEditor
- ColorEditor
- CompoundEditor
- DirectoryEditor
- EnumEditor
- FileEditor
- FontEditor
- HTMLEditor
- ImageEnumEditor
- InstanceEditor
- ListEditor
- RGBColorEditor
- RangeEditor
- SetEditor
- TableEditor

- TextEditor
- TitleEditor
- TreeEditor
- TupleEditor

3.2 Advanced Demos

- Adapted TreeEditor
- Apply/Revert Handler
- Table (read-only, auto-edit table column)
- TabularEditor (auto-update)
- DateEditor
- Dynamic EnumEditor
- Dynamic Range Editor
- Dynamic Views
- HDF5 Tree
- History
- Invalid state handling
- ListStrEditor
- ListEditor
- ListEditor (notebook tabs)
- MVC
- Multi-selection string list
- Multithread
- Multithread 2
- NumPy array TabularEditor
- NumPy ArrayViewEditor
- Popup Dialog
- Property List
- ScrubberEditor
- Settable cached property
- Statusbar
- StringListEditor
- TableEditor (with checkbox column)
- TableEditor (with context menu)
- TableEditor (with live search and cell)

- TabularEditor
- TimeEditor

CHAPTER 4

Indices and tables

- genindex
- search

A

ApplyButton, 15
attribute, **97**
attributes
    Group, 10
    Item, 6
    View, 16

B

buttons
    attribute, 14
    examples, 15
    standard, 15

C

CancelButton, 15
class attribute, **97**
close_result attribute, 18
columns attribute, 10
command button, **97**
configure_traits()
    default view example, 19
    examples, 4
    method, 22
    view parameter, 5, 21
content attribute
    Group, 10
    View, 18
context, 23
    examples, 23
    View, 18
control, 6
controller, 2, **97**
Controller (built-in class), 27

D

default view, 19, 20
    example, 19
default_traits_view()
    default view method, 20
defined_when attribute
    Group, 12
    Item, 6
dialog box, **97**
dock attribute
    Group, 10
    Item, 6
    View, 18
dock_theme attribute, 10
drop_class attribute, 18

E

edit_traits(), 22
editor, **97**
editor attribute, 6
editor factory, **97**
emphasized attribute, 6
enabled_when attribute
    Group, 12
    Item, 6
environment variable
    ETS_TOOLKIT, 3
ETS_TOOLKIT, 3
ETSConfig.toolkit, 3
examples
    buttons, 15
    configure_traits(), 4
    context, 23
    default view, 19
    Include, 25
    multi-object Views, 23
    multiple Views, 21
    View object, 5
export attribute
    Group, 10
    Item, 6
    View, 18
extended trait names
    Item name attribute, 24
extended_traitname_changed(), 29

F

factory, **97**
format_func attribute, 6
format_str attribute, 6

G

Group, **97**
    attributes, 10
    subclasses, 12
group_theme attribute, 10

H

Handler, **97**
handler attribute, 18
Handler class
    as MVC controller, 2
has_focus attribute, 6
HasTraits, **97**
HasTraits class
    as MVC model, 2
Heading class, 8
height attribute
    Item, 6
    View, 18
help attribute
    Group, 12
    Item, 6
    View, 18
help_id attribute
    Group, 12
    Item, 6
    View, 18
HFlow, 13
HGroup, 13
HSplit, 13

I

icon attribute, 18
id attribute
    Group, 12
    Item, 6
    View, 18
image attribute
    Group, 10
    Item, 6
    View, 18
imports attribute, 18
Include
    examples, 25
    object, 25
instance, **97**
Item, **97**
    attributes, 6
    object, 6
    subclasses, 8
item_theme attribute
    Group, 10
    Item, 6
    View, 18

K

key_bindings attribute, 18
kind attribute, 13

L

label attribute
    Group, 10
    Item, 6
Label class, 8
label_theme attribute
    Group, 10
    Item, 6
    View, 18
layout attribute, 10
live, **97**
    definition, 14
    window kind, 13
LiveButtons, 16
livemodal, **97**
livemodal window kind, 13

M

menubar attribute, 18
modal, **97**
    definition, 14
    window kind, 13
ModalButtons, 16
model, 2, **98**
Model-View-Controller, 2
model_view attribute, 18
ModelView (built-in class), 27
multi-object Views, 23
    examples, 23
multiple Views, 21
    examples, 21
MVC, **97**
MVC design pattern, 2

N

name attribute, 6
NoButtons, 16
nonmodal, **98**
nonmodal window kind, 13
null toolkit, 3

O

object, **98**

Include, 25
Item, 6
View, 4, 5
object attribute
Group, 10
View, 18
OKButton, 15
OKCancelsButtons, 16
on_apply attribute, 18
orientation attribute, 10

P

padding attribute
Group, 10
Item, 6
panel, 14, **98**
window kind, 13
predefined trait type, **98**

Q

Qt toolkit, 3

R

resizable attribute, 6
View, 18
RevertButton, 15

S

scrollable attribute, 18
selected attribute, 10
show_border attribute, 10
show_label attribute, 6
show_labels attribute, 10
show_left attribute, 10
Spring class, 8
springy attribute
Group, 10
Item, 6
statusbar attribute, 18
style attribute
Group, 10
Item, 6
View, 18
subclasses
Group, 12
Item, 8
subpanel, 14, **98**
window kind, 13

T

Tabbed, 13
title attribute, 18
toolbar attribute, 18
toolkit
environment variable, 3
flag, 3
selection, 3
tooltip attribute, 6
trait, **98**
trait attribute, **98**
trait type, **98**
Traits, **98**
traits_view attribute, 21
TraitsUI, **98**

U

ui(), 22
UndoButton, 15
updated attribute, 18

V

VFlow, 13
VFold, 13
VGrid, 13
VGroup, 13
View, **98**
as MVC view, 2
attributes, 16
contents, 5
context, 18
customizing, 13
default, 19
external, 18, 22
internal, 18
methods for displaying, 22
multi-object, 23
multiple, 21
object, 4, 5
ways of displaying, 18
view, **98**
view (in MVC), 2
ViewElement, **98**
visible_when attribute
Group, 12
Item, 6
VSplit, 13

W

widget, 6, **98**
width attribute
Item, 6
View, 18
windows
panel, 14
stand-alone, 14
subpanel, 14
wizard, 14
wizard, 14, **98**

window kind, 13
wx, **98**
wxPython toolkit, 3

X

x attribute, 18

Y

y attribute, 18